
Common Design Patterns for Symbian OS

The Foundations of Smartphone Software

Lead Author

Adrian Issott

With

**Nicholas Addo, Toby Gray, David Harper, Craig Heath,
Guruprasad Kini, Ian McDowall, Ben Morris, John Roe,
Dale Self, James Steele, Jo Stichbury, Oliver Stuart, Viki Turner,
Hamish Willee**

Reviewed by

**Sandip Ahluwalia, Matthew Allen, Sorin Basca, David Caabiero,
Iain Campbell, Douglas Feather, Ryan Gilmour,
Martin Hardman, Richard Harrison, Tim Howes, Mark Jacobs,
Martin Jakl, Sian James, Antti Juustila, Simon Mellor,
Sihem Merah, Will Palmer, Subhasis Panigrahi, Lucian Piros,
Mark Shackman, Adrian Taylor, Paul Todd, Gabor Torok,
Charles Weir, Alex Wilbur, Tim Williams**

Head of Symbian Press

Freddie Gjertsen

Managing Editor

Satu McNabb



A John Wiley and Sons, Ltd., Publication

Common Design Patterns for Symbian OS

The Foundations of Smartphone Software

Common Design Patterns for Symbian OS

The Foundations of Smartphone Software

Lead Author

Adrian Issott

With

**Nicholas Addo, Toby Gray, David Harper, Craig Heath,
Guruprasad Kini, Ian McDowall, Ben Morris, John Roe,
Dale Self, James Steele, Jo Stichbury, Oliver Stuart, Viki Turner,
Hamish Willee**

Reviewed by

**Sandip Ahluwalia, Matthew Allen, Sorin Basca, David Caabiero,
Iain Campbell, Douglas Feather, Ryan Gilmour,
Martin Hardman, Richard Harrison, Tim Howes, Mark Jacobs,
Martin Jakl, Sian James, Antti Juustila, Simon Mellor,
Sihem Merah, Will Palmer, Subhasis Panigrahi, Lucian Piros,
Mark Shackman, Adrian Taylor, Paul Todd, Gabor Torok,
Charles Weir, Alex Wilbur, Tim Williams**

Head of Symbian Press

Freddie Gjertsen

Managing Editor

Satu McNabb



A John Wiley and Sons, Ltd., Publication

Copyright © 2008

John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester,
West Sussex PO19 8SQ, England

Telephone (+44) 1243 779777

Email (for orders and customer service enquiries): cs-books@wiley.co.uk

Visit our Home Page on www.wileyurope.com or www.wiley.com

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except under the terms of the Copyright, Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London W1T 4LP, UK, without the permission in writing of the Publisher. Requests to the Publisher should be addressed to the Permissions Department, John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex PO19 8SQ, England, or emailed to permreq@wiley.co.uk, or faxed to (+44) 1243 770620.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The Publisher is not associated with any product or vendor mentioned in this book.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold on the understanding that the Publisher is not engaged in rendering professional services. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 6045 Freemont Blvd, Mississauga, Ontario, L5R 4J3, Canada

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 978-0-470-51635-5

Typeset in 10/12 Optima by Laserwords Private Limited, Chennai, India

Printed and bound in Great Britain by Bell & Bain, Glasgow

This book is printed on acid-free paper responsibly manufactured from sustainable forestry in which at least two trees are planted for each one used for paper production.

Dedicated to Kat and to Mike

Contents

Author Biographies	ix
Authors' Acknowledgments	xv
Foreword	xvii
Glossary	xix
1 Introduction	1
1.1 About this Book	1
1.2 Who this Book Is For	1
1.3 Which Version of Symbian OS this Book Is For	2
1.4 General Design Patterns	2
1.5 Symbian OS Patterns	4
1.6 Design Pattern Template	10
1.7 Structure of this Book	12
1.8 Conventions	13
1.9 Other Sources of Information	13
2 Error-Handling Strategies	15
Fail Fast	17
Escalate Errors	32
3 Resource Lifetimes	49
Immortal	53

Lazy Allocation	63
Lazy De-allocation	73
4 Event-Driven Programming	87
Event Mixin	93
Request Completion	104
Publish and Subscribe	114
5 Cooperative Multitasking	131
Active Objects	133
Asynchronous Controller	148
6 Providing Services	165
Client-Thread Service	171
Client-Server	182
Coordinator	211
7 Security	233
Secure Agent	240
Buckle	252
Quarantine	260
Cradle	273
8 Optimizing Execution Time	287
Episodes	289
Data Press	309
9 Mapping Well-Known Patterns onto Symbian OS	331
Model-View-Controller	332
Singleton	346
Adapter	372
Handle-Body	385
Appendix: Impact Analysis of Recurring Consequences	397
References	403
Index	407

Author Biographies

Nicholas Addo

Nicholas has worked on Symbian OS for six years. He first joined the Browser Technology group in Symbian, developing a generic framework for content handling. He then moved to the Personal Information Management group to work on Enterprise Group Scheduling technology. From there, he joined the Multi-technology development group to work on cross-technology and process-improvement projects. This included a stint in product management defining a Component Technology product strategy for Symbian OS and an ongoing involvement in improvement initiatives for Symbian OS core idioms.

Nicholas started his career as an apprentice Electronics Engineer in the 1980s, developing semiconductor devices for telecommunications. He moved into a solely software development career in the 1990s. This was initially in real-time embedded systems and later Windows-based user interfaces for image scanners, writers, and print register and color control devices for the printing and pre-press industry.

Toby Gray

Toby has worked with Symbian OS since joining the System Management group of Symbian in 2005. Since then, he has worked on topics ranging from optimizing system characteristics (e.g. speed, RAM usage and ROM usage) to management of system start-up and from development of diagnostic tools to ARM assembler optimization. He holds a BA in Computer Science from the University of Cambridge.

David Harper

David has worked at Psion Software and Symbian for over 10 years. During that time he has developed an in-depth knowledge of HTTP, Internet technologies, ECom and Platform Security. He now works in Integration Management, specializing in Technology Integration. David holds an MSc in Artificial Intelligence from Queen Mary and Westfield College.

Craig Heath

Craig has been working in IT security since 1988, with The Santa Cruz Operation as security architect for SCO UNIX, then at Lutris Technologies as security architect for their Java Enterprise Application Server. He joined Symbian in 2002, working in product management and strategy. He has contributed to several published industry standards and has a long association with The Open Group Security Forum, including co-authorship of the Open Group Technical Guide to Security Design Patterns. Craig is lead author of *Symbian OS Platform Security*.

Adrian Issott

Adrian joined Symbian's Shortlink team in 2004 working first on developing support for Bluetooth stereo headsets and then on enhancing the Bluetooth HAI. In 2006, he moved on to become an architect focusing on system characteristics such as performance, RAM usage, reliability and security. His time is currently spent either working on long-term architectural improvements in areas such as system start-up and trace or on urgent projects helping device manufacturers optimize their devices and get the best out of Symbian OS. Adrian graduated from Jesus College, Oxford with a first class honors degree in Mathematics.

Guruprasad Kini

Guru has been working with Symbian for the last three and a half years, most of which he spent in the Remote Management Team. He is now a part of the Systems Engineering Team. Design patterns, cryptography and data security have been his primary professional interests. Guru is based near Bangalore and holds a BE in Computer Science from Manipal Institute of Technology, Karnataka. He hopes to retire early and do what he always wanted to do – teach mathematics in schools. And . . . oh yes, he is also very bad at writing bios.

Ian McDowall

Ian joined Symbian in 2001, has worked in a range of teams within Symbian and is currently a Technology Architect responsible for Shortlink technologies. He has previously filled roles ranging from developer through project manager to technical manager. He has an MA in Computer Sciences from Cambridge University and an MBA from Warwick University. As a software engineer for over 25 years, he has worked for a number of software companies and has worked on more than 15 operating systems, developing software ranging from enterprise systems to embedded software. He is married to Lorraine and they have two children, Kelly and Ross, and a number of pets.

Ben Morris

Ben joined Psion Software in October 1997, working in the software development kit team on the production of the first C++ and Java SDKs for what was at that time still the EPOC32 operating system. He led the small team that produced the SDKs for the ER5 release of EPOC32 and, when Psion Software became Symbian, he took over responsibility for expanding and leading the company's system documentation team. In 2002, he joined the newly formed System Management Group in the Software Engineering organization of Symbian, with a brief to 'define the system'. He devised the original system model for Symbian OS and currently leads the team responsible for its maintenance and evolution. He can be found on the Internet at www.benmorris.eu.

John Roe

John has an MA in Engineering from Cambridge University and has worked with Symbian for well over 10 years. For the past nine years, he has been in Symbian's Customer Engineering group as the S60 Technical Lead. He has worked on S60 since its inception and on many S60 phones ranging from the Nokia 7650 to present-day devices. He divides his time between supporting Symbian's licensees and using his product realization experience to develop the architecture of Symbian OS.

Dale Self

Dale is soon to celebrate 10 years of working for Symbian, where he worked first in the Messaging team on IMAP and later in the Shortlink

team on Bluetooth and OBEX. Most recently, he has been working on USB architecture and standardization. In his spare time, he enjoys music, photography and writing fiction as well as non-fiction. He hopes that his contribution to this book falls into the latter category.

James Steele

James joined Symbian after graduating from university in 2004. He attended the University of Cambridge (Fitzwilliam College) where he read Computer Science. Since starting his career at Symbian as a member of the Shortlink team he has been involved with the development of numerous protocol stacks, including Bluetooth, USB and IrDA.

In his free time, James enjoys taking part in friendly poker matches with friends and colleagues, and pretending to be a film critic. He plays various sports, including basketball, badminton and squash. In the winter, you will probably find him skiing somewhere in the French Alps.

Jo Stichbury

Jo is Senior Technical Editor with Symbian Press. She has worked within the Symbian ecosystem since 1997 in the Base, Connectivity and Security teams of Symbian, as well as for Advansys, Sony Ericsson and Nokia. Jo is the author of *Symbian OS Explained: Effective C++ programming for smartphones*, published by Symbian Press in 2004. She also coauthored *The Accredited Symbian Developer Primer: Fundamentals of Symbian OS* with Mark Jacobs, in 2006. Her most recent book is *Games on Symbian OS: A handbook for mobile development*, published in early 2008.

Oliver Stuart

Oliver is a recent convert to Symbian and joined the company's Comms team in 2007. Thrown in at the deep end, he has enjoyed the challenging work of developing for a powerful mobile operating system. He previously worked in the field of biological digital imaging at Improvision Ltd and has interests in object-oriented programming, concurrency, networking and multimedia. He holds a BSc with Hons in Computer Science/Software Engineering from the University of Birmingham in the UK. He has carefully tried to hide his past as a Windows developer from his new colleagues to avoid derision.

Viki Turner

Viki is the Comms Chief Technology Architect at Symbian. She has worked with communications technologies at Symbian since joining in 2001, when she helped the third Symbian phone to ship successfully. In earlier engineering roles, she also developed communications middleware – first with an interesting but short-lived startup, developing a networked multimedia distribution platform, and then working in the City of London on data feed products. Viki trained in computing at Imperial College, after a few years pursuing a career as a classical soprano . . . but that's another story.

Hamish Willee

Hamish joined Symbian in late 1998 and is looking forward to the 10 year celebrations! Most of those years were spent in developer support roles – he still gets a kick out of improving third-party development on Symbian OS. The last few years he has worked remotely from Melbourne, Australia, continuing to provide technical assistance to Symbian's global developer ecosystem. Outside work, Hamish loves to spend time with his family: Jen, Oscar, Sam and Leo.

Authors' Acknowledgments

We would like to thank:

- The individual authors who worked long and hard not only to describe the patterns but also to refine the key ideas that needed to be expressed.
- All the technical reviewers who corrected and enhanced the text.
- Jo Stichbury who helped at every step of the way with insightful advice on how to improve the book as well as authoring a pattern herself.
- Satu McNabb who kept us focused on delivering to our deadlines.
- Charles Weir who helped to ensure the patterns don't just document how Symbian OS is architected but are accessible to developers building on Symbian OS. His positive attitude was also very welcome!
- Everyone else in Symbian who helped us along the way especially Ian McDowall, John Roe, Matthew Reynolds and Michael Harper.
- Our friends and families who, throughout the long process of preparing this book, didn't see as much of us as they had the right to expect.

Foreword

This book gives you an insight into the world's most successful smartphone operating system. As this book goes to press, over 200 million Symbian smartphones have shipped worldwide – twice as many as all other types of smartphone put together. It's a huge potential market for applications and add-in software.

Symbian OS is a very powerful environment; writing the most effective software for it means learning the Symbian dialect of C++ and thinking in the idioms it uses. Who better to teach that than some of the software architects who designed the operating system in the first place? Adrian and his team have produced a book that teaches you to 'think' in Symbian OS. Other books can teach you the mechanics of the language – how to use the GUI, the networking APIs and the application infrastructure – but this book helps you to develop a feel for the way a complex application should fit together, which is much more difficult.

My own involvement with Symbian OS included being technical architect for the first Symbian OS phone, co-authoring a patterns book on limited memory software, and running Penrillian, a company specializing in porting software to Symbian OS. All these roles have emphasized for me just how much mobile phone development differs from traditional desktop and server programming. Symbian OS is very strong in two features: effective use of power and avoiding resource leaks when handling error conditions. Teaching developers the design techniques to keep software conforming to these has been a major task for me; these patterns make it straightforward.

This work builds on the introductions to Symbian OS in other Symbian Press books. Even if you already have a good understanding of how to use leaves and traps, active objects, GUI controls and the communications APIs, you'll need to know how to design effective architectures and

component interfaces to use them effectively. That's why this book is so valuable. It describes the thinking of the architects who designed Symbian OS, its applications and infrastructure, and tells us, in bite-sized pieces, how they did it so that we can build on their work. As a seasoned Symbian OS programmer, I learnt something new from the descriptions of managing secure plug-ins and the process-coordination patterns. If you don't have much experience on Symbian OS, you will particularly value the explanation and idioms of inter-process communication, resource management and event handling.

How does the pattern format help you to learn? A book of patterns is half way between a descriptive volume and a reference text. You can read each pattern as a separate paper, but the prescriptive pattern format also makes it easy to skim through all the patterns and see which ones cover which ground. So please don't think of this as a book to read from cover-to-cover (though you can do that if you like). Instead you can use the way it's set out to skim through the overviews of each pattern: the names, intent, and descriptions – and read as much as you like of each of the patterns you find interesting or use a lot already. As you work on Symbian OS projects and designs, you'll find yourself with problems that remind you of one or more of the solutions in the book – and that is the time to go back and look for the detail of the implementation. The book becomes the experienced designer leaning over your shoulder saying 'oh yes, have a look at such-and-such for the way to do it, and don't forget thus-and-this issues to consider'. And it's experienced designers that make architecture great.

So take a look. No matter who you are, I challenge you not to find a technique – or an implementation of a technique – that's interesting to you. And you'll find, as I did, that you can learn a lot about how best to design Symbian OS applications and services. Here's wishing you success designing software for the most popular mobile platform in the world!

Charles Weir
Penrillian, 2008

Glossary

Word	Definition
ABI	Application Binary Interface.
AO	Active Object.
Application developer	A person who develops applications for Symbian OS that run on top of a specific UI Layer. Often such applications do not form part of a ROM and are installed onto a device after it has been created in a factory.
Base port provider	A company that provides the base port required to get Symbian OS working on specific device hardware.
Binary compatibility	This is less restrictive than full compatibility in that it states only that a client executable does not need to be re-compiled for it to work with all versions of the binary compatible executable.
Compatibility	Software is said to be compatible if the clients of one version of the software can also be run on all other versions of the software without any changes being needed to the clients.
Component	A block of code that performs a particular function at an 'interesting' level of functionality.
CPM	Communication provider module.
Device creator	A developer of software that is integrated into a ROM during device manufacture. Device creators most commonly work for Symbian, a UI vendor or a base port provider but could also be one of their suppliers.
DLL loading rule	A process can only load a DLL if that DLL has been trusted with at least the capabilities that the process has.

Word	Definition
DRM	Digital rights management (DRM) refers to technologies used by publishers and copyright holders to control access to and usage of digital media.
Ephemeral resources	Resources such as CPU cycles and battery power are ephemeral. Their key trait is that you can't hold on to them or reserve them for your own use. They're there but you can't keep a grip on them whether you use them or not.
Executable	A file whose contents are meant to be interpreted as a program by a computer. Not just EXE files but DLLs and other file types as well.
Framework	A framework provides a reusable design for a software system or subsystem. It is expressed as a set of abstract classes and the way their instances collaborate, with the intent of facilitating software development by allowing developers to spend more time on meeting software requirements rather than dealing with the more tedious low-level details of providing a working system. Before a framework can be deployed it needs to be specialized for a particular use. This can either be done at build time via static dependencies between software modules or at run time via dynamically loaded plug-ins.
ICL	Image Conversion Library, part of the Symbian OS multimedia subsystem.
IPC	Inter-Process Communication.
LDD	Logical Device Driver.
Mixin	An abstract interface that provides a certain functionality to be inherited by a class. The Symbian OS naming convention for these classes is that they start with an 'M'.
MMU	Memory Management Unit.
Non-XIP	A device which is not XIP-based stores code on disk, usually compressed in some way, and loads it into RAM to be run. This is in addition to the RAM used for the run-time data for each instance. Non-XIP devices based on Symbian OS are currently the most common type outside Japan.
Ownable resources	Resources such as RAM, disk space and communication channels are ownable. Unlike ephemeral resources, you can hold onto them and keep using them. Such resources can be reserved for your sole use.
PDD	Physical Device Driver.
Plug-in	A software module loaded at run time to add a specific feature or service to a larger software module.
Polling	This is where a software object periodically requests information. This is the opposite of event-driven programming, in which the software object would wait to be notified of any changes to the information it is interested in.

Word	Definition
Race condition	This is a flaw in a system or process whereby the output or result of the process is unexpectedly and critically dependent on the sequence or timing of other events. The term originates with the idea of two signals racing each other to influence the output first.
RAM	Random Access Memory.
Real time	This is a constraint on an operation in which the time between starting and finishing it must be within a fixed limit.
Resource	Any physical or virtual entity of limited availability that needs to be shared across the system.
SDL	This is the Symbian Developer Library which can be found in every device SDK as well as online at developer.symbian.com/main/documentation/sdl .
Source compatibility	This is less restrictive than full compatibility as it states only that no source code changes need to be made for a client to work across all versions of the software.
Stray signal	This is when a request has been signalled as complete but either there is no active object available or an active object is found that has neither been set active nor has a request pending.
SWI	Software Installer.
TCB	Trusted Computing Base can refer to a capability or to the components that use the capability, such as the kernel, the file server and, on open devices, the software installer. These components have unrestricted access to the device resources. They are responsible for maintaining the integrity of the device and applying the fundamental rules of platform security. The rest of the operating system must trust these components to behave correctly and as a result, their code will have been very strictly reviewed.
TCE	Beyond the TCB, other system components are granted orthogonal restricted system capabilities and constitute the Trusted Computing Environment. This includes servers providing sockets, telephony, certificate management, database access, and windowing services. Each server has just the capabilities it requires; for instance, the windowing server is not granted phone stack access and the communications server is not granted direct access to keyboard events.
TCP	Transmission Control Protocol.
UDP	User Datagram Protocol.
UI vendor	A company that provides the user interface layers on top of Symbian OS which, along with a base port, is needed to create platforms from which devices can be created. The current UI layers are MOAP, S60 and UIQ.

Word	Definition
Vendor	A company or person that provides software for use on a Symbian OS device.
WSD	Writable Static Data is any modifiable globally-scoped variable declared outside of a function, or any function-scoped static variables.
XIP	An eXecute-In-Place device is one in which code is stored and run directly from disk. This means that the code does not take up any RAM when run although each instance still uses RAM for its run-time data, such as its program stack, etc. XIP devices based on Symbian OS are currently the most common type in Japan.

1

Introduction

1.1 About this Book

If you've ever asked yourself 'How do the experts architect software for mobile devices?' then this book is for you. *Common Design Patterns for Symbian OS* collects the wisdom and experience of some of Symbian's finest software engineers. It distils their knowledge into a set of common design patterns that you can use when creating software for Symbian smartphones.

This book helps you negotiate the obstacles often found when working on a smartphone platform. Knowing the potential problems, and the patterns used to solve them, will give you a head start in designing and implementing robust and efficient applications and services on Symbian OS.

This book is not intended to be a quick start guide to Symbian OS, nor to explain its design or architecture. Other titles from Symbian Press do just that, for example [Harrison and Shackman, 2007] and [Morris, 2007]. Rather, it aims to capture expert knowledge about programming practice specific to Symbian OS and make it available to all developers working with Symbian OS.

We welcome your feedback and contributions and so we have provided a wiki page at [**developer.symbian.com/design-patterns-wiki**](http://developer.symbian.com/design-patterns-wiki) where you can post your experiences with the patterns we describe and share additional patterns that you know of for Symbian OS.

1.2 Who this Book Is For

This book is suitable for relative beginners as well as experts in Symbian OS development. It will help beginners to make progress quickly when working on medium-sized projects. It will also support experts in their design of large-scale and sophisticated software and assist them in learning

from the experience of other experts. The book should help both groups to harness well-proven solutions, as well as specific variations, to individual design problems.

To get the most out of the book, readers are expected to have some existing knowledge of Symbian OS and C++, so basic concepts are not explained in detail. For instance, we assume that you are aware of some of the basic idioms, such as descriptors, and have an understanding of the basic types of Symbian OS classes (e.g. C, R, T and M classes) and how they behave. The book specifically targets existing Symbian C++ developers whether they're creating applications or services. It can equally well be used by developers internal to Symbian, licensee developers creating devices and third-party developers writing after-market applications.

If you are new to Symbian OS or want to take a refresher course in these concepts, there are several books that you could read, including [Harrison and Shackman, 2007] and [Stichbury and Jacobs, 2006], which describe the basics of Symbian C++ development.

1.3 Which Version of Symbian OS this Book Is For

This book applies to Symbian OS v9.1 and onwards, which means that all the patterns described work in all v9.x releases. However, a number of the patterns are not specific to any particular version of Symbian OS and can therefore be applied to earlier versions of the operating system as well.

The patterns contained in this book are also expected to be equally applicable to any future version of the Symbian platform. While some of the details of the operating system may well change, it is unlikely that there will be significant differences that impact the level of abstraction in which the patterns of this book are described.

1.4 General Design Patterns

The genre of software design pattern books was established with the publication of the 'Gang of Four' book [Gamma *et al.*, 1994] in the 1990s. Since then a profusion of design pattern books have been published, including a small number explicitly for embedded systems, of which [Weir and Noble, 2000] is a good example.

The aim of all of these books is to capture the collective experience of skilled software engineers based on proven examples so as to help promote good design practices. This is achieved through the creation of

design patterns, commonly defined to be ‘solutions to a problem in a context’. The process of creating a design pattern reflects the way that experts, not just in software engineering but in many other domains as well, solve problems. In most cases, an expert will draw on their experiences of solving a similar problem and reuse the same strategies to resolve the problem at hand. Only rarely will a problem be solved in an entirely unique fashion. Design patterns are established by looking for common elements in the solutions to similar problems.

The Introduction to [Gamma *et al.*, 1994] and the Patterns chapter of [Buschmann *et al.*, 1996] provide a good discussion of pattern theory so we instead focus on how to get the best out of design patterns. Most people start with a design problem that they’re not sure how to solve and so might start flipping through a design patterns book like this one looking for inspiration. Each of the patterns is written in such a way that it should be as easy as possible to quickly understand whether it’s describing a solution to the problem you’re concerned with and whether it’ll help you construct software with the properties you require. Once you’ve identified a pattern that looks like it should help, you should take some time to customize it for your specific circumstances. Ideally, using a pattern is comparable to approaching a problem as if you’ve solved something similar in a previous project. Then when you’re reviewing, implementing or documenting your software using a well-defined pattern gives you a common vocabulary to help communicate to other engineers involved in the project how your software is designed.

Here are some important things to keep in mind when using patterns:

- All patterns have negative consequences in addition to the positive reasons that lead you to consider a given pattern in the first place. These negatives also need to be taken into account when choosing whether or not to use a pattern.
- You don’t get points for using as many patterns as possible when solving a single problem. Patterns should only be used when they are actually needed. The main reason for this is that using too many can result in overly complicated solutions caused by the additional levels of abstraction that they generally introduce.
- Patterns are not intended to be used as fixed templates and applied directly to your software without thinking. Instead they should be used as a guide to get you most of the way towards describing a working solution. It’s up to you to finish off the design process and ensure that your specific solution respects the details of your situation.

Patterns are often very useful tools but only when applied to the right jobs.

1.5 Symbian OS Patterns

That's all very well but what's this got to do with Symbian OS? Well, one problem with patterns is that, because they have been abstracted away from specific problems, it can be hard to understand how best to apply them. One of the goals of this book is to describe a set of patterns in more tangible terms specific to Symbian OS and so make them easier to apply in a Symbian OS context. This will be done via the following means:

- Each pattern describes how the forces most relevant to an embedded operating system, such as Symbian OS, impact and are changed by the patterns.
- Explanations are given to explain how Symbian OS architectural elements can be re-used in the application of the patterns.
- Code samples are given in terms of Symbian OS APIs and coding standards so that as much as possible of your job of translating a pattern into practice has already been done for you.
- Examples based on Symbian OS are given to show how the patterns have been realized in situations that Symbian OS developers can more easily relate to.

1.5.1 Constraints on Software Based on Symbian OS

The constraints on a mobile device platform means that software development for Symbian OS has to resolve a set of forces that are distinct from those found in desktop or enterprise environments. Consider the following list of constraints imposed by the mobile device environment:

- *Constrained hardware*
Compared to mainstream laptops, mobile devices are between 10 and 30 times smaller in terms of CPU speed, RAM size and storage space and are set to remain so in relative terms even as they increase in absolute terms. This is partly to do with the fact that mobile devices are expected to be carried around in your pocket all the time, putting both size and weight constraints on the devices. Another factor is the fact that with over 7 million Symbian OS devices now being sold each month,¹ device manufacturers are keen to maximize their profits by keeping the cost of the materials in each device as low as possible. Increasing the instructions processed per second means including a faster CPU; boosting the memory available to the system means including a bigger RAM chip. When shipping potentially millions of each product, even small savings per device add up quickly. Also, if an end user finds that a device frequently runs

¹www.symbian.com/news/pr/2008/pr20089781.html.

out of resources, they can't just plug in a new bit of hardware as they can on a PC.

- *Limited power supply*
Unlike fixed desktop computers or enterprise servers, mobile devices have to run from a battery. This limited power supply has to be carefully conserved so that the device can stay active for as long as possible, especially because battery life is one of the top requirements of end users. This constraint means that peripherals such as antennas, RAM chips, CPU, the screen, and so on, should all be used as little as possible or used at a reduced rate to save power.
- *Expectations of high reliability*
End users expect and demand rock-solid stability and reliability from their mobile devices; desktop levels of quality are not acceptable. One of the drivers for this is the fact that mobile devices can be a life-saving tool whether it's to make an emergency phone call or to allow the rescue services to locate you if you've had an accident. An additional factor is that it is relatively easy for an end user to switch between mobile devices; any devices that don't measure up to this standard are returned to the manufacturer.
- *Open operating system*
A key difference between Symbian OS and many other embedded devices is that the operating system allows the software on a device to be extended and upgraded after it has been shipped from the factory and is in the hands of an end user. Hence, software cannot be optimized for specific tasks or rely on events occurring in a set order. Software based on Symbian OS needs to be flexible, adaptable and allow for future changes.
Note that whilst upgrades are possible on a mobile device, they're usually restricted to functionality upgrades rather than providing defect fixes. This is because getting the upgrade onto the device may well take longer and may cost the user money if it is sent over the phone network. Frequent updates would also increase the chance of mistakes occurring and hence affect the reliability of the device unless carefully controlled.
- *Expectations of high security*
All of the stakeholders in mobile devices – users, device manufacturers, retail vendors, network operators, and suppliers of software, services, and content – expect them to be secure against software attacks. The risks include end users losing their private data, content producers having their DRM files stolen and damage to the reputation of device manufacturers. The risks to mobile devices are significant because Symbian OS allows software to be installed: there are multiple communication channels to and from the device, they

often contain lots of private information, and they support a number of services that can incur charges through operator tariffs, premium-rate messages, and so on.

- *Restricted user interface*

Due to the small size of mobile devices not only are the screen sizes small but they also suffer from minimal keyboards, if a keyboard is present at all, and other restricted human interface mechanisms. This means that user interfaces have to be made simpler. In addition, they need to be more adaptable than in larger devices because the end user is more likely to make an error in interacting with the device. Note that this constraint mainly affects applications rather than the underlying services.

1.5.2 Important Forces in the Context of Symbian OS

The constraints listed in Section 1.5.1 lead to the following important forces for software based on Symbian OS:

- *Minimal use of RAM*

Random Access Memory (RAM) is a type of computer data storage. A key characteristic of RAM is that it is volatile and information it contains is lost when a device is turned off. The three main contributors to RAM are:

- Call stack – a dynamic data structure which stores information about the current execution including details of the active functions and local data
- Heap – an area of memory that you can dynamically allocate from using the `new` operator; ‘free store’ is another name for a heap
- Code – devices that are not eXecute-In-Place (XIP) based load code from secondary storage into RAM where it is executed. Exactly how this is done depends on the memory model used but code can make up a significant proportion of the RAM used on a device.

The main motivation for reducing RAM is simply because there is a limited amount available on the device. However, a secondary consideration is that power is required to store data in RAM so reducing RAM can also result in less power usage. On the other hand, minimizing RAM usage can be at the cost of increasing the execution time if, for example, you reduce cache sizes.

- *Predictable RAM usage*

A necessary consequence of having limited RAM available is that at some point it will run out. For some situations this can have catastrophic consequences for your device. For instance, if an end user is trying to call for an ambulance, the phone shouldn’t fail because the

software couldn't allocate enough memory. More commonly however, being able to predict memory usage means that your software can prepare in advance and deal with the side-effects of allocating RAM outside of the critical path. This allows you to deal with any errors as well as the time needed for the allocations without disrupting your main functionality. One trade-off that you'll probably have to make to improve this force is to use more RAM in situations where you over-predict the actual RAM usage.

- *Minimal use of secondary storage*

Secondary storage is where code, read-only data and persistent data are stored. Most Symbian OS devices support secondary storage by providing an area of flash memory² though there exist some that have a hard disk drive.

In all mobile devices, the amount of secondary storage is limited. You also need to remember that accessing it is much slower than accessing RAM. Another factor to consider is that secondary storage suffers from wearing and degrades as it is used, though this is not often a significant factor when writing software unless you're a device creator.

- *Minimal execution time*

The execution time for a piece of software can have a number of meanings. Usually you are concerned about a particular use case, whether it's starting an application or responding to a key press from the end user, and wish to measure the time taken between that use case starting and ending. You can also start taking into account the amount of time your software was active during the use case. However, the main concern is to make your use case execute in as short a time as possible. A use case is almost always constrained by a bottleneck. This bottleneck may well be the CPU but it could just as well be some other hardware constraint, such as reading from secondary storage or network bandwidth.

Reducing execution time is good for the end user's experience with the device but you may also reduce power usage by simply doing less work, whether by sending fewer instructions to the CPU or through reduced use of the network. On the other hand, you may have to increase your RAM usage to achieve this, for instance by adding a cache to your application or service.

- *Real-time responsiveness*

For some types of software being fast is not the most important objective; instead they have to operate within strict deadlines.³ When developing such software you need to consider all of the execution

²en.wikipedia.org/wiki/Flash_memory.

³Of course, if you're unlucky your software will need both to be fast and to meet time constraints!

paths through your software, even the exceptional cases, to ensure that not only are they all time-bounded but that they complete before the specified deadline.

One downside to optimizing for this force is that you may increase the development cost and reduce the maintainability of your component by introducing an artificial separation between your real-time and non-real-time operations.

- *Reliability*

This is the ability of your software to successfully perform its required functions for a specified period of time. The reliability of your software reflects the number of defects that your software contains and so is one measure of the quality of your application or service. The drawbacks of ensuring your software is reliable are mainly the increased effort and cost of development and testing.

A major part of ensuring that your software is reliable on a mobile device is designing in ways of handling sudden power loss gracefully. The expectation of an end user is that they will be able to continue seamlessly from where they left off once they've restored power to the device in some way. As power can be lost at any time and your software is probably given very little warning, this means the normal activity of your component must take this into account by, for instance, regularly saving data to disk.

- *Security*

Secure software protects itself and the stakeholders of a device from harm and prevents security attacks from succeeding. The key difference between security and reliability is that secure software must take into account the actions of active malicious agents attempting to cause damage. However, increased security, like reliability, can result in increased effort and cost of development and testing. Another possible disadvantage is reduced usability since, almost by definition, increasing security means reducing the number of valid actions that a client can perform.

1.5.3 Other Forces

The forces mentioned in Section 1.5.2 are particularly important on a mobile device but we shouldn't forget some of the more standard forces on software designs:

- *Development effort*

Software development is the translation of the need of an end user or marketing goal into a product. You should aim to design your software to reduce the total effort required to achieve this. This is particularly important at the architecture and design stages as the choices you make early on in a project will have a big impact further down the line

when you are implementing, testing and maintaining your software. Nonetheless you do need to ensure you spend sufficient time at each stage of development to get these benefits.

- *Testing cost*

Software testing is the process used to assess the quality of your software. It is important to remember that, for most practical applications, testing does not completely establish the correctness of a component;⁴ rather, it gives you a level of confidence in how reliable your software will be. With this in mind, it is important to design your software so that testing is made as easy as possible to allow you to quickly and easily establish the assurance you need to ship your software.

- *Maintainability*

This is defined by [ISO 9126] to be ‘the ease with which a software product can be modified in order to correct defects, meet new requirements, make future maintenance easier or cope with a changed environment’. Before you invest too heavily in this, hence increasing your development effort, you need to be sure that your product will benefit from it. For instance, if you only wish to target UIQ devices then making your application UI highly portable isn’t necessary. Of course that assumes you don’t change your mind later!

- *Encapsulation*

This is the ability to provide the minimum necessary well-defined interface to a software object in such a way that the internal workings are hidden. A well-encapsulated object will appear to be a ‘black box’ for other objects that interact with it without needing to know how it works. The goal is to build self-contained objects that can be plugged into and out of other objects cleanly and without side effects.

Improving the encapsulation of your objects helps with reliability because objects do what you expect them to do. It also helps with maintainability because you minimize the risk that extending or changing the software breaks other parts of the application or service. It also reduces testing cost by allowing you to more easily isolate an object for unit testing. However, this is at the expense of reduced flexibility.

1.5.4 Pattern Elements that Are Already in Place

The patterns in this book are specified in terms of Symbian OS and describe how to reuse any of the utility libraries or services provided as part of the operating system so that you can get your solution up and running as quickly as possible.

⁴Correctness is difficult to establish due to the extremely large number of possible input states and the correspondingly large number of execution paths through a piece of software. Hence 100% test coverage is virtually impossible to obtain.

In some cases, the solution presented in a pattern describes how to use an existing framework to help resolve your design problem. This is to be expected since patterns promote design reuse and, hence, in some cases, code is also reused. One example of this is the *Active Objects* pattern (see page 133) where Symbian OS provides a lot of machinery to get you going whilst giving you the flexibility to apply the pattern to your situation.

1.6 Design Pattern Template

Each pattern in this book is described using the following template:

Pattern Name

Intent A one-line description of the pattern to give you a rough idea of the aim of the design pattern.

AKA Stands for 'also known as' and gives other common names for the pattern.

Problem

Context

A single sentence that describes the environment or situations in which the problem occurs. This section provides a clear outline of the background to the problem and the assumptions that this entails.

Summary

Bullet points summarize each of the main forces involved in the problem that the pattern has to address. This section gives an at-a-glance answer to the question 'is this pattern going to be suitable for my particular problem?'

Description

Here, longer paragraphs expand on the bullet points in the summary. They answer questions such as 'when and where might this pattern apply?', 'is this a problem only when creating services or when building applications?', 'what makes this a particularly difficult problem?' and 'might each particular type of Symbian OS developer (device creator, application developer, etc.) experience this problem or do they just need an understanding of the pattern?' In most cases, a pattern

will be fully implementable by all types of developers. In all cases, at least portions of the pattern will implementable by any Symbian OS developer.

Example

A real-world example of a problem based on Symbian OS that demonstrates the existence of the pattern.

Solution

One or two paragraphs summarizing the fundamental solution principle underlying the pattern.

Structure

This section gives a detailed specification of the structural aspects of the pattern showing the classes involved, their responsibilities and what they collaborate with.

Dynamics

This section describes the important scenarios that explain the run-time behavior of the pattern using object-messaging sequence charts.

Implementation

This section gives practical guidelines for implementing the pattern. These are only a suggestion, not an immutable rule. You should adapt the implementation to meet your needs, by adding different, extra, or more detailed steps, or, sometimes, by re-ordering the steps. Symbian OS C++ code fragments are given to illustrate a possible implementation.

Consequences

This section lists the typical positive and negative outcomes resulting from the use of the pattern.

Example Resolved

The example described in the Problem section of this pattern is resolved in this section using the pattern. Often code samples are given to illustrate the key points along with a detailed discussion of how the example works and explanations of how it has been designed. This includes a discussion

of any important aspects for resolving the specific example that are not covered by the analysis elsewhere in the pattern.

Other Known Uses

One or more brief descriptions of additional real-world examples, based on Symbian OS, are given here of the use of the pattern. Often these are uses originating from within Symbian OS since that's the area we best understand, though we have also included examples drawn from external sources. These examples are not described in great detail, only enough to outline how the pattern has been used.

Variants and Extensions

A brief description of similar designs or specializations of the pattern.

References

References to other patterns that solve similar problems as well as to patterns that help us refine the pattern we are describing.

1.7 Structure of this Book

The patterns in this book cover the following topics:

- Error-handling strategies: How to handle errors effectively so that their impact on the end user is minimized
- Resource lifetimes: How to work effectively with the constrained resources available to a Symbian OS smartphone
- Event-driven programming: How to conserve power whilst reducing the coupling between your components
- Cooperative multitasking: How to take advantage of Symbian's cooperative multitasking framework and get the appearance of multithreading without the associated costs
- Providing services: How to make your functionality available to multiple clients either individually or concurrently
- Security: How to use the platform security architecture to secure your own application and services
- Optimizing execution time: How to increase the speed with which your software executes and reduce the time it takes to start up

- Mapping well-known patterns onto Symbian OS: How to apply well-known design patterns, such as Adapter, Singleton and Model–View–Controller, on Symbian OS

In each case, whether you're a device creator or an application developer, you'll find all of these patterns help you to write software that better harnesses the unique characteristics of mobile devices.

1.8 Conventions

To help you get the most out of this book and keep track of what's happening, we've used a number of typographical conventions throughout:

- When we refer to words used in code, such as variables, classes and functions, or to filenames, then we use this style: `iEikonEnv`, `ConstructL()`, and `e32base.h`.
- When we list code, or the contents of files, we use the following convention:

```
This is example code;
```

- URLs are written like this: ***www.symbian.com/developer***.
- Classes or objects that form part of Symbian OS are shown in a darker color to indicate that they should be reused when a pattern is implemented. Classes or objects that you are expected to implement yourself are shown in a lighter color.

1.9 Other Sources of Information

Several times in this book we refer to the Symbian Developer Library (SDL). This is available online at ***developer.symbian.com/main/oslibrary*** and is integrated into the documentation that comes with the S60 and UIQ SDKs, which can also be found online on Forum Nokia (for S60) at ***www.forum.nokia.com*** and (for UIQ) on the UIQ Developer Community website at ***developer.uiq.com***.

The official website for this book can be found at ***developer.symbian.com/design-patterns.book***. You can download sample code, read an interview with the lead author and download an electronic copy of the first chapter. The sample code provided on this site will be a fuller version of the code snippets given in the implementation sections of selected patterns. Whilst this sample code will not be from a real-world example,

it will be illustrative of the associated pattern particularly because it'll be complete enough to be run. However, don't forget that this code should not be simply be cut and pasted into your production code, not least because patterns are not strict templates but rather guidelines that should be modified to suit your specific environment.

You can find links to related articles and sites with additional information about patterns on the wiki page at ***developer.symbian.com/design_patterns_wikipage***. The wiki can also be used to send us feedback about the book or even to submit your own favorite design patterns for Symbian OS.

2

Error-Handling Strategies

This chapter deals with patterns for handling errors appropriately so that their impact is mitigated and dealt with effectively. We first need to define the categories of software errors that can occur:

- *Domain errors* are associated with a particular technology. They indicate that a problem that can be expected to happen at some point in time has occurred. An example of this type of error is getting `KErrNotFound` as a result of a multimedia application trying to open an audio file specified by the end user. The failure to find the file has a specific meaning for the software receiving the error.
- *System errors* are the result of attempting some operation that fails due to a restriction of the system. `KErrNoMemory` is an obvious example, since it occurs whenever a memory allocation fails due to the memory constraints imposed by the system.
- *Faults* occur because of programming errors. They result in abnormal conditions that cause a failure to execute the desired functionality of a particular system. Examples include null pointer dereferences and out-of-bound array accesses. User applications, core system services and the kernel can all suffer faults because of the same common factor: the human engineer.

Some other useful terms we should define here are:

- *Defect* – also known as a bug, these are flaws, mistakes and failures in software that prevent it from behaving as intended. All faults are defects but not all defects are faults. For example, software could be said to be defective if it has been not been implemented according to the original requirements specified. It is also important to note that domain and system errors are not defects in themselves, because they result from circumstances outside the control of your software. However, the handling of these kinds of errors could result in a defect.

- *Resolving an error* – an action is taken at run time within a software component to return the execution back to normal operation. For instance, this may be as simple as retrying an operation or providing a useful message to the end user to report what has occurred.
- *Handling an error* – an error is either resolved or deliberately passed on for another component to resolve.

Defects in software are a reality and, no matter what we do, they can never be completely eliminated. As the size and complexity of a system increases, so does the potential number of defects present. If we are careless in designing a system, then this increase in defects can be exponential since the number of defects grows in line with the number of interactions in the system which itself grows exponentially as the number of components increases. One of the reasons for using interfaces and patterns such as Façade [Gamma *et al.*, 1994] is that they reduce the number of different interactions. Another thing to remember is that any modification made to software has a probability of introducing new defects which means that defect fixes can give rise to yet more defects.

When targeting mobile devices, such as Symbian OS smartphones, there are some specific issues that need to be taken into account when designing software:

- End users have expectations of high reliability from their devices which means error handling should be considered carefully when designing software for Symbian OS. Mobile phones are consumer electronics and are considered to be completely reliable. They are not expected to behave like computers, which are forgiven the need for an occasional re-boot, but are expected to function perfectly whenever they are used.
- A significant proportion of mobile phone users are not very technically literate and so do not expect to have to fix software problems themselves or try to understand cryptic error dialogs.
- Patching your software is possible on most Symbian smartphones but should be avoided where possible since it's likely to cost the end user to download the new version over the network.

The first pattern in this chapter, *Fail Fast* (see page 17), helps you reduce the number of faults that are present in your released software and minimize the impact that they have when they do occur. The second pattern, *Escalate Errors* (see page 32), describes how you can ensure that an error is resolved cleanly and in the most appropriate manner.

Fail Fast

Intent Improve the maintainability and reliability of your software by calling `Panic()` to deal with programming errors as soon as they are detected.

AKA Programming by Contract [Meyer, 1992]

Problem

Context

Detecting faults in your software should be done as early as possible during the development process since it can be over 10 times cheaper to detect them in the early stages of development compared to finding them once your product has shipped [Fagan, 1976].

Summary

- You want to reduce the number of defects in your software and so improve the experience for the end user by, for example, avoiding loss of their privacy, corruption of their data, and security vulnerabilities.
- You want to reduce the effort associated with debugging and fixing defects.
- You want to improve the maintainability of your software by making it easier to add new features safely.

Description

The problem we're focusing on here is how to detect faults in our software since a common issue when developing software of any complexity is programmer error. These mistakes manifest themselves as defects, some of which will be very apparent while developing a component; others are much more subtle and therefore long-lived and insidious. Although they are all faults, there is a key difference between defects found during the production of a system by its developers and those discovered by end users of the software.

Finding defects during the development of a system has significantly less cost¹ than when they are found after the system has been deployed.

¹This cost is in both the engineering time to investigate and find the root cause of the defect and in deploying the fix. In addition, there is a less tangible cost associated with damage to the reputation of the software vendor. If the error were to cause devices to be returned to the manufacturer then the costs could run into millions of dollars.

Therefore a lot of programming effort should go into producing code that is functionally correct and as error free as possible; however, the effort required to reach perfection steadily increases.² While simple, small, standalone programs can be debugged relatively trivially, discovering the root cause of a defect in larger, more complex systems can prove very time-intensive. The reason for this general trend is that there is a greater number of components in larger systems, with an exponentially greater possible number of interactions between them. This is one reason why software engineers are encouraged to abstract, encapsulate and modularize their designs, since it reduces the number of these interactions (as promoted by patterns such as Proxy and Façade as described in [Gamma *et al.*, 1994]).

The root cause for a defect can be hard to track down in any system:

- It may be that the defect doesn't occur in a development environment and it is only out in the field where the issue is first discovered. This may be for a number of reasons, such as race conditions caused by event-driven programming or simply because there weren't enough tests written.
- The design of the software may not be well understood by the developer investigating a defect.
- The more complex a system becomes, the more likely it is that the defect symptom manifests itself in a 'distant' component in which there is no obvious connection between the observed issue and the root cause of the defect.

Such defects can have a significant maintenance cost associated with them. Notably they will be found later in the software lifecycle and will take longer to fix.

For software based on Symbian OS, reducing the occurrence of faults is especially important when compared to desktop (or even enterprise) systems. Issuing software updates (or patches) is a common practice for desktop PCs that are permanently connected to the Internet, with large amounts of available storage and bandwidth. Patching software on a mobile device is less widespread, more complex and can be more expensive due to data transmission costs. Reducing the need to issue incremental updates to correct defects is of great value. Less tangibly, there is also the user's expectation of reliability. Given the current climate, end users are used to their PCs containing defects and the requirement for anti-virus software, firewalls, and continual patching is accepted. For Symbian OS though, devices are generally expected to work, out of the box, and be always available. To meet these expectations, removal of defects early in the development cycle is essential.

²en.wikipedia.org/wiki/Diminishing_Returns.

Example

An example of a complex software component would be a multiplexing protocol. For such a component there are three different viewpoints from which to observe its software requirements:

- API calls from clients on top of the stack – these direct the protocol to perform tasks, such as who to connect to and to send or receive data.
- Internal state – the constructs used by the software to satisfy API requests from clients while respecting the protocol specification for communicating with remote devices.
- Protocol messages from a remote device – data and control packets sent as part of the communication protocol both to and from peers.

An example of such a protocol is the Bluetooth Audio Video Distribution Transport protocol (AVDTP) which has a specified interface known as the Generic Audio Video Distribution Profile (GAVDP).³

A GAVDP client will have a number of requirements on the API. These requirements will normally be mapped onto the features laid out in the GAVDP specification published by the Bluetooth Special Interest Group (SIG) which includes connecting to a remote device, discovering remote audio–video (AV) stream endpoints, determining the capabilities of an endpoint, as well as configuring and controlling a logical AV stream. This is in addition to the fundamental requirement of sending and receiving the AV data associated with an AV stream.

The protocol implementation must conform to the specification defined by the Bluetooth SIG and, as is often the case with protocol specifications, it is important to handle all the (sometimes vast numbers of) possible cases and interactions that are permitted by the specification. The net result is that a fairly complex state machine is required to manage valid requests and responses from remote devices, while also robustly handling domain errors (such as invalid transactions from defective or even malicious devices), system errors (such as failing to allocate enough memory), and faults (such as typing errors in hard-coded constants).

In addition, there is the logic to map the API and the protocol together. Although initially this may appear to be fairly straightforward, for an open operating system this is rarely the case. There can be multiple, distinct GAVDP clients using the protocol to communicate with multiple devices, or even the same device. The stack is required to co-ordinate these requests and responses in a robust and efficient manner.

We hope to have convinced you that the Symbian OS GAVDP/AVDTP protocol implementation is a complex component of software. It is apparent that faults could occur locally in a number of places: from incorrect

³For more information see [Campbell *et al.*, 2007, Chapter 4] and www.bluetooth.com/Bluetooth/Technology/Works.

usage of the API by clients, from lower layers corrupting messages, and from mistakes in the complex logic used to manage the protocol.

As with all software based on Symbian OS, it is paramount that there are minimal faults in the released component. In this case, the consequences of faults can be serious ranging from jitter in the AV stream preventing end users from enjoying the content to allowing DRM-protected data to be sent to an unauthorized device.

Solution

The basic principle of this solution is to 'panic' – terminate the current thread of execution – as soon as an unexpected condition (i.e. a fault) arises, rather than using an inappropriate default or trying to ignore the event and carrying on regardless.

The reason for panicking is to prevent the thread from attempting to do anything more and allowing the symptoms of the fault to spread. In addition, it provides a convenient debugging point at which a call stack, representing an execution snapshot, can be retrieved. In debug mode, a panic can trigger a breakpoint and allow you to enter your debugger.

This pattern explicitly encodes design constraints in software and checks that they are being met. This prevents the scope of a problem growing by restricting the issue to a single thread rather than risking the entire device. This could be considered as forming the foundation of a fault-tolerant system.

Structure

This pattern focuses on the concrete places within a software component where you can add lines of code, known as *assertions* or, more colloquially, as *asserts*, where a check is performed that the design constraints for your component are being met. It is when an assert fails that the current thread is panicked.

We classify asserts into two different types:

- *External asserts* check for the design constraints imposed on how software outside the component interacts with it. An example would be clients of an API provided by the component. If an external assert fails then it indicates that the client has used the component incorrectly. You should be able to test that these asserts fail during testing as they effectively form part of the API itself.
- *Internal asserts* check for the design constraints imposed on the component itself. If an internal assert fails then it indicates that there is a fault in the component that needs to be fixed. The unit tests for the component should seek to test that these asserts cannot be caused to fail.

Figure 2.1 illustrates how these two types of asserts are used to validate different aspects of the design.

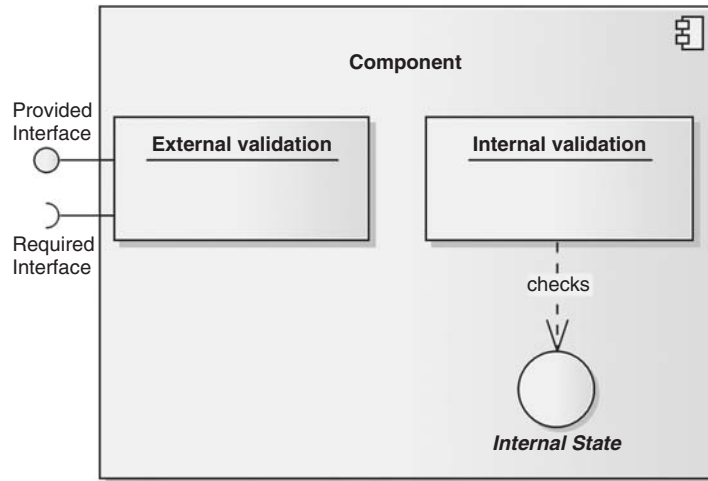


Figure 2.1 Structure of the *Fail Fast* pattern

Some concrete examples of where you might add asserts are:

- within the implementation of a public interface
- when a transition is made in a state machine so that only valid state changes are performed
- checking a class, or even a component, invariant within functions (an invariant is a statement that can be made about the class or component that should remain true irrespective of what operations you perform on it).

Of course, there are a number of situations in which you would not wish to assert but which you would instead handle in a more sophisticated manner. One such case is that of expected unexpected errors. In plain English this is the set of errors that should have been considered (by design), but whose arrival can occur unexpectedly at any time. Often this type of error is either a system or a domain error. A good example of this is the disconnection of a Bluetooth link, since it can be disconnected at any time by a request from a remote device, noise on 'the air', or by moving the devices out of radio range.

Another case that typically should not be externally asserted is incorrect requests from a client that are sensitive to some state of which the client is not aware. For instance, a client calling `Read()` on a communication socket before `Connect()` has been called is a state-sensitive request that can be asserted since the client should be aware of the socket's

state.⁴ However, you should not assert based on state from one client when handling the request from another. This sounds obvious but is often much less so in practice, especially if you have a state machine that can be manipulated by multiple clients who know nothing of each other.

Dynamics

The sequence chart in Figure 2.2 illustrates how a class function could implement a rigorous scheme of asserts. The function initially checks that the parameters passed in are suitable in a pre-condition checking step as well as verifying that the object is in an appropriate state to handle the function call. The function then performs its main operation before executing a post-condition assert that ensures a suitable output for the function has been computed, before again checking that the object has not violated any of its invariants.

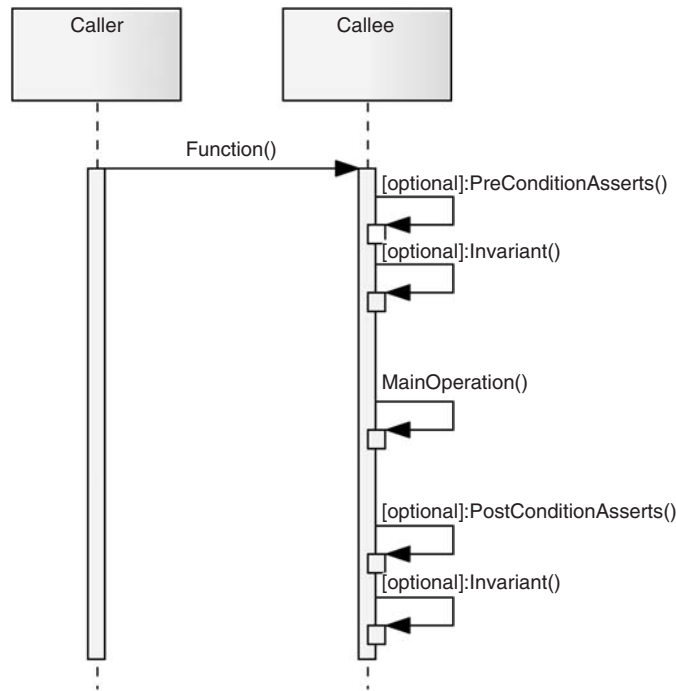


Figure 2.2 Dynamics of the *Fail Fast* pattern

It's not essential that all functions conform to this sequence; the diagram simply provides a general example of how a function could implement this pattern.

⁴You should consider carefully before adding external state-sensitive asserts to an interface because they place additional burdens on clients that use it.

By using this pattern to check the design contracts internal to a component we have effectively produced a form of unit-test code built into the component. However, these test steps are in a form that needs to be driven as part of the standard testing for the component. This pattern and testing are complementary since testing is used to show that the implementation of a component's design is behaving correctly (shown by the asserts) whilst the asserts aid the debugging of any design violations (shown by the testing).

Implementation

Reducing the Impact of the Asserts

A naïve interpretation of this pattern would simply be to always check as many conditions as are necessary to validate the implementation of the design. Whilst for some types of software this may be appropriate, it is often crucial to take into account the various issues that this approach would introduce.

The two main ways that asserts impact your software is through the additional code they add to an executable and the additional execution time needed to evaluate them. A single assert might only add a handful of bytes of code and take nanoseconds to evaluate. However, across a component, or a system, there is potentially a significant number of asserts, especially if every function implements pre- and post-condition checking. For Symbian OS development, it is often simply too expensive to use asserts liberally in production software and so we need to be selective in where they are added.

One approach to this is to always enforce external asserts because we have no control over client code and what they will try to do. However, the same is normally not true for internal asserts as these should *never* be triggered if the software is operating correctly and hence a trade-off can be made between the performance cost of asserts and the benefits they bring. The simplest solution is to ensure that internal asserts are only present in debug code. The assumption is that you are continually running your test suite on both the debug and release versions of your component and hence can have a reasonable level of confidence that the any new faults introduced into the component will be identified during testing. However, the consequences of this solution are that there will always be some release-only faults that will be let through by this method and will not be stopped by an assert.

A more sophisticated approach to dealing with the internal asserts is to change which builds they are compiled into during the development process. Initially you would choose to have them compiled into both release and debug builds until you have confidence⁵ that enough faults

⁵This may come from some quantifiable metric such as code coverage.

have been removed from your component, at which point you could leave the asserts only in the debug builds.

Implementing a Single Assert

Symbian OS provides the following standard assert macros in `e32def.h`:

- `__ASSERT_ALWAYS(c, p)`
Here `c` is a conditional expression which results in true or false and `p` is a statement which is executed if the conditional expression `c` is false.
- `__ASSERT_DEBUG(c, p)`
The same as `__ASSERT_ALWAYS(c, p)` except that it is compiled out in release builds.
- `ASSERT(c)`
The same as `__ASSERT_DEBUG(c, p)` except that it causes a USER 0 panic if `c` is false. Using this standard Symbian OS macro is no longer recommended since there's no way of identifying which `ASSERT()` in a component caused a USER 0. Hence, it is common for components to redefine the macro to provide more details that allow the asserts to be distinguished during debugging.
- `__ASSERT_COMPILE(c)`
This macro asserts that `c` is true at compilation time and is particularly useful for checking hard-coded constants although the error messages it causes the compiler to output could be more informative.

In addition, you can use the following macros to help check class invariants in a consistent manner:

- `__DECLARE_TEST`
This should be added as the last item in a class declaration so that a function called `__DbgTestInvariant()` is declared. It is your responsibility to implement the invariant checks and call `User::Invariant()` to panic the thread if they fail.
- `__TEST_INVARIANT`
This calls the `__DbgTestInvariant()` function in debug builds.

For more information on these macros, please see the Symbian Developer Library.

However, you still need to know how to cause a panic if the assert fails. A panic is most commonly issued on the local thread, by calling `User::Panic(const TDesC& aCategory, TInt aReason)` which is declared in `e32std.h`. The category is a textual value,⁶ and the

⁶At most `KMaxExitCategoryName` or 12 characters long.

reason is a numeric value; together they form a description of the cause of a panic. This description is shown on the screen in a dialog box as well as being sent to the debug output. For instance:

```
_LIT(KPanicCategory, "Fail Fast");
enum TPanicCode
{
    EInvalidParameter,
    EInvalidState,
    EInappropriateCondition,
    ...
};
void CClass::Function(const TDesC& aParam)
{
    __ASSERT_ALWAYS(aParam.Length() == KValidLength,
        User::Panic(KPanicCategory, EInvalidParameter));
    __ASSERT_DEBUG(iState == ESomeValidState,
        User::Panic(KPanicCategory, EInvalidState));

    // Function implementation
}
```

Using the parameters passed to `Panic()` in a disciplined way provides useful debugging information. Notably, if the category and reason uniquely map to the assert that caused the panic then even if a stack trace or trace data for the fault is not available⁷ then someone investigating the fault should still be able to identify the condition that caused the problem. For external asserts, time spent explicitly creating a unique panic category and reason combination for every assert in the component is often time well spent. Taking the example above, `EInvalidParameter` could become `EInvalidParameterLengthForCClassFunction` and `EInvalidState` may become `EInvalidStateForCallingCClassFunction`, and so on.

One of the reasons for explicitly setting the category and reason for external asserts is that they form part of the API for clients of your component. Developers using the API will expect consistency not only for its run-time behavior but also in how they debug their client code, for which identifying a particular panic is key. The use of external asserts helps to maintain compatibility for the API of which they form a part. By documenting and enforcing the requirements for requests made on an interface more rigidly (by failing fast), it becomes easier to change implementations later as it is clear that clients must have adhered to the specific criteria enforced by asserts.

One problem with this is that the development cost of explicitly assigning the category and reason for each separate panic is proportional to the number of asserts and so can become time consuming. An alternative that is well suited to internal asserts is to have the panic category assigned

⁷As can be the case for those rare issues that are only seen on a release device.

automatically as the most significant part of the filename and the reason as the line number:

```
#define DECL_ASSERT_FILE(s) _LIT(KPanicFileName,s)
#define ASSERT_PANIC(l) User::Panic(KPanicFileName()).
    Right(KMaxExitCategoryName),l)
#define ASSERT(x) { DECL_ASSERT_FILE(__FILE__);
    __ASSERT_ALWAYS(x, ASSERT_PANIC(__LINE__)); }
```

This does have one big disadvantage which is that you need to have the exact version of the source code for the software being executed to be able to work out which assert caused a panic since the auto-generated reason is sensitive to code churn in the file. Not only might the developer seeing the panic not have the source code, even if he does the person attempting to fix the problem will probably have difficulty tracking down which version of the file was being used at the time the fault was discovered. However, for internal asserts that you don't expect to be seen except during development of a component this shouldn't be a problem.

Panicking the Correct Thread

It is critical to fail the appropriate entity when using this pattern. For internal asserts, it is not necessarily an issue since it is nearly always the local thread. However, for external asserts policing requests from clients, it is not so straightforward:

- In libraries, either statically or dynamically linked, it is the current thread.
- In services residing in their own process, it is the remote thread that made the request.

For the latter case, when using *Client–Server* (see page 182), the client thread can be panicked using the `RMessagePtr2::Panic()` function:

```
void CExampleSession::ServiceL(const RMessage2& aMessage)
{
    ...

    if(InappropriateCondition())
    {
        aMessage.Panic(KPanicCategory, EInappropriateCondition);
        return;
    }

    ...
}
```

Alternatively you can use `RThread::Panic()` to panic a single thread or `RProcess::Panic()` to panic a process and all of its threads.

Ultimately the choice of where and how to fail fast requires consideration of users of the software and some common sense.

Consequences

Positives

- The software quality is improved since more faults are found before the software is shipped. Those faults that do still occur will have a reduced impact because they're stopped before their symptoms, such as a hung application or corrupted data, increase.
- The cost of debugging and fixing issues is reduced because of the extra information provided by panic categories and reasons, in addition to problems being simpler because they're stopped before they cause knock-on problems.
- The maintainability of your component is improved because the asserts document the design constraints inherent in its construction.
- Security is improved because faults are more likely to stop the thread executing than to allow arbitrary code execution.

Negatives

- Security can be compromised by introducing denial-of-service attacks since a carelessly placed assert can be exploited to bring down a thread.⁸
- Carelessly placed external asserts can reduce the usability of an API.
- Code size is increased by any asserts left in a release build. On non-XIP devices, this means increased RAM usage as well as additional disk space needed for the code.
- Execution performance is impaired by the additional checks required by the asserts left in a release build.

Example Resolved

The Symbian OS GAVDP/AVDTP implementation applies this pattern in several forms. It is worth noting that the following examples do not constitute the complete usage of the pattern; they are merely a small set of concise examples.

API Guards

The GAVDP API uses the Fail Fast approach to ensure that a client uses the API correctly. The most basic form of this is where the API ensures

⁸Or even the whole device, if the thread is system critical.

that the `RGavdp` object has been opened before any further operations are attempted:

```
EXPORT_C void RGavdp::Connect(const TBTDevAddr& aRemoteAddr)
{
    __ASSERT_ALWAYS(iGavdpImp, Panic(EGavdpNotOpen));
    iGavdpImp->Connect(aRemoteAddr);
}
```

The implementation⁹ goes further to police the API usage by clients to ensure that particular functions are called at the appropriate time:

```
void CGavdp::Connect(const TBTDevAddr& aRemoteAddr)
{
    __ASSERT_ALWAYS((iState == EIdle || iState == EListening),
        Panic(EGavdpBadState));
    __ASSERT_ALWAYS(iNumSEPsRegistered, Panic(
        EGavdpSEPMustBeRegisteredBeforeConnect));
    __ASSERT_ALWAYS(aRemoteAddr != TBTDevAddr(0),
        Panic(EGavdpBadRemoteAddress));
    __ASSERT_ALWAYS(!iRequesterHelper, Panic(EGavdpBadState));
    ...
}
```

Note that the above code has state-sensitive external asserts which, as has been mentioned, should be carefully considered. It is appropriate in this particular case because the `RGavdp` class must be used in conjunction with the `MGavdpUser` class,¹⁰ whose callbacks ensure that the client has sufficient information about whether a particular function call is appropriate or not.

Invariant, Pre- and Post-condition Checking

A lot of code written in Symbian OS uses asserts to check design constraints and AVDTP is no exception. For performance reasons, the majority of these asserts are only active in debug builds:

```
void CAvdtpProtocol::DoStartAvdtpListeningL()
{
    LOG_FUNC
    // Check that we haven't already got an iListener.
    // NOTE: in production code we will leak an iListener.
```

⁹Note that the `RGavdp` and `CGavdp` classes are the handle and the body as a result of the use of the *Handle–Body* pattern (see page 385) in their design.

¹⁰See *Event Mixin* (page 93).

```
// These are fairly small so not too severe.
__ASSERT_DEBUG(!iListener, Panic(EAvdtpStartedListeningAgain));

...
}
```

The example shows a good practice: using a comment to explicitly state what will happen in release builds if the particular fault occurs. In the code above, although we do not want to leak memory, the debug assert combined with comprehensive testing should give us confidence that this condition will never actually arise.

The use of asserts is not limited to complex functions. Even simple functions can, and should, check for exceptional conditions:

```
void CManagedLogicalChannel::ProvideSAP(CServProviderBase* aSAP)
{
    __ASSERT_DEBUG(aSAP, Panic(EAvdtpPassingNullSapOwnershipToChannel));
    __ASSERT_DEBUG(!iLogicalChannelSAP, Panic(
        EAvdtpPassingSapOwnershipToChannelThatAlreadyHasASap));
    iLogicalChannelSAP = aSAP;
    iLogicalChannelSAP->SetNotify(this);
}
```

The above example shows the use of a pre-condition check, firstly that the parameter to the function is not NULL, and secondly that the object upon which the function is called does not already have a Service Access Point (SAP) bound to it.

These last two example code snippets demonstrate the use of panic reasons that are unique across the whole component by design and probably across the whole of Symbian OS through the use of the EAvdtp prefix. Although the names of the particular panic reason enumeration values can be fairly long, they are self-documenting and thus no accompanying comment is required. Furthermore, they provide a good demonstration of how asserts can document the design constraints of an implementation.¹¹

State Transition Checking

The checking of state transitions can be thought of as a special case of invariant checking. The following function is the base class implementation of the 'Set Configuration' event for an abstract class representing

¹¹From the `ProvideSAP()` function, it is apparent that the `CManagedLogicalChannel` class is only ever intended to have a single logical channel (represented by a SAP) bound to it and that it takes ownership of the SAP passed into the function.

an audio–video stream state as per the State pattern [Gamma *et al.*, 1994]:

```
void TAVStreamState::SetConfigurationL(
    CAVStream& /*aStream*/,
    RBuf8& /*aPacketBuffer*/,
    CSignallingChannel& /*aSignallingChannel*/,
    TBool /*aReportingConfigured*/,
    TBool /*aRecoveryConfigured*/) const
{
    LOG_FUNC_DEBUGPANICINSTATE(EAvdtpUnexpectedSetConfigurationEvent);
    User::Leave(KErrNotReady);
}
```

In this particular example, we can see the base class implementation triggers an assert if in debug mode, but in a release build the fault will be handled as an exceptional error. The reason for this is that State classes, derived from `TAVStreamState`, which are by design expected to receive ‘Set Configuration’ events must explicitly override the `SetConfigurationL()` function to provide an implementation to deal with the event. Failing to provide a way to handle the implementation is considered to be a fault.

Other Known Uses

This pattern is used widely throughout Symbian code and here we give just a select few:

- *Descriptors*
Symbian OS descriptors panic when an out-of-bound access is attempted. The prevalent use of descriptors for buffers and strings in Symbian OS means that the possibility of arbitrary code execution through a buffer overflow attack against Symbian OS components is vastly reduced. This type of panic is an external panic, as it forms part of the API to a bounded region of memory.
- *Active Objects*
The active scheduler panics if it detects a stray signal rather than just ignoring it as it indicates that there is an error with an asynchronous request made via an active object. By panicking as soon as this condition is detected, debugging the defect is much easier than trying to track down why some notifications do not occur occasionally.
- *Symbian Remote Control Framework*
This component auto-generates the panic reason for its internal asserts which are compiled only into debug builds. These asserts are liberally used throughout the component.

Variants and Extensions

- *Passing Additional Debug Information when Panicking*
The usual procedure for assigning the category and reason for a panic is to give the component name for the category and then just assign a fixed number for the reason. However, the category can be up to 16 letters and the reason is 32 bits so there are usually opportunities for additional information to be given here. For instance the `ALLOC` panic is used when a memory leak has been detected and the reason contains the address of the memory. Also, some state machines that raise panics use the bottom 16 bits of the reason to indicate which assert caused the problem and the top 16 bits to indicate the state that it was in at the time.
- *Invoking the Debugger when Failing an Assert*
Symbian OS provides the `__DEBUGGER()` macro which can be used to invoke a debugger as if you had manually inserted a breakpoint where the macro is used. This does not result in the thread being killed. The debugger is only invoked on the emulator and if 'Just In Time' debugging has been enabled for the executing process.¹² In all other circumstances, nothing happens.
This variant combines the use of this macro with `__ASSERT_DEBUG()` calls but otherwise applies the pattern described above. This allows you to inspect the program state when the assert failed and so better understand why it happened.

References

- *Escalate Errors* (see page 32) is related because it may be useful for those domain and system errors for which this pattern cannot be used.
- *Code that tests itself: Using conditions and invariants to debug your code* [Weir, 1998].
- [en.wikipedia.org/wiki/Assertion\(computing\)](http://en.wikipedia.org/wiki/Assertion(computing)) is an article on the general use of assertions when developing software.

¹²It is enabled by default but can be changed using `User::SetJustInTime()` or `RProcess::SetJustInTime()`.

Escalate Errors

Intent Enhance your error handling by using the Symbian OS Leave mechanism to escalate the responsibility of handling an error up to the point which has enough context to resolve the error.

AKA Leave and Trap, Error Propagation, Exception Handling, and Separating Error-Handling Code from ‘Regular’ Code

Problem

Context

Your component is part of or contains a layered software stack¹³ and has to appropriately handle non-fatal domain or system errors.

Summary

- Non-fatal errors must be resolved appropriately so that an application or service can continue to provide the functionality the user expects.
- There should be a clear separation between code for error handling and the normal flow of execution to reduce development time and maintenance costs by reducing code complexity and increasing software clarity.
- Error-handling code should not have a significant impact at run time during normal operation.

Description

Most well-written software is layered so that high-level abstractions depend on low-level abstractions. For instance, a browser interacts with an end user through its UI code at the highest levels of abstraction. Each time the end user requests a new web page, that request is given to the HTTP subsystem which in turn hands the request to the IP stack at a lower level. At any stage, an error might be encountered that needs to be dealt with appropriately. If the error is fatal in some way, then there’s little you can do but panic as per *Fail Fast* (see page 17).

Non-fatal error conditions, however, are more likely to be encountered during the operation of your application or service and your code needs to be able to handle them appropriately. Such errors can

¹³As per the Layers pattern described in [Buschmann *et al.*, 1996] or *Model–View–Controller* (see page 332).

occur at any point in the software stack whether you are an application developer or a device creator. If your software is not written to handle errors or handles them inappropriately¹⁴ then the user experience will be poor.

As an illustration, the following example of bad code handles a system error by ignoring it and trying to continue anyway. Similarly it handles a recoverable missing file domain error by panicking, which causes the application to terminate with no opportunity to save the user's data or to try an alternative file. Even if you try to set things up so that the file is always there, this is very easily broken by accident:

```
void CEngine::Construct(const TDesC& aFileName)
{
    RFs fs;
    fs.Connect(); // Bad code - ignores possible system error

    RFile file;
    err = file.Open(fs,aFileName,EFileRead); // Could return KErrNotFound
    ASSERT(err == KErrNone);
    ...
}
```

In general, it is not possible for code running at the lowest layers of your application or service to resolve domain or system errors. Consider a function written to open a file. If the open fails, should the function retry the operation? Not if the user entered the filename incorrectly but if the file is located on a remote file system off the device somewhere which does not have guaranteed availability then it would probably be worth retrying. However, the function cannot tell which situation it is in because its level of abstraction is too low. The point you should take away from this is that it may not be possible for code at one layer to take action to handle an error by itself because it lacks the context present at higher layers within the software.

When an error occurs that cannot be resolved at the layer in which it occurred the only option is to return it up the call stack until it reaches a layer that does have the context necessary to resolve the error. For example, if the engine of an application encounters a disk full error when trying to save the user's data, it is not able to start deleting files to make space without consulting the end user. So instead it escalates the error upwards to the UI layer so that the end user can be informed.

It would be inappropriate to use *Fail Fast* (see page 17) to resolve such errors by panicking. Whilst it does have the advantage of resolving the current error condition in a way which ensures that the integrity of data stored on the phone is unlikely to be compromised, it is too severe a

¹⁴Such as by using *Fail Fast* (see page 17) when there was a possibility of a safe recovery.

reaction¹⁵ to system or domain errors that should be expected to occur at some point.

Unfortunately, the C++ function call mechanism provides no distinct support for error propagation by itself, which encourages the development of ad-hoc solutions such as returning an error code from a function. All callers of the function then need to check this return code to see if it is an error or a valid return, which can clutter up their code considerably.

As an example of where this is no problem to solve, consider the error-handling logic in the TCP network protocol. The TCP specification requires a peer to resend a packet if it detects that one has been dropped. This is so that application code does not have to deal with the unreliability of networks such as Ethernet. Since the protocol has all the information it needs to resolve the error in the layer in which it occurred, no propagation is required.

Example

An example of the problem we wish to solve is an application that transmits data via UDP using the Communication Infrastructure subsystem¹⁶ of Symbian OS, colloquially known as *Comms-Infras*. The application is likely to be split up into at least two layers with the higher or UI layer responsible for dealing with the end user and the lower or engine layer responsible for the communications channel.

The engine layer accesses the UDP protocol via the `RSocket` API but how should the engine layer handle the errors that it will receive from this API? How should it handle errors which occur during an attempt to establish a connection?

- Clearly it should take steps to protect its own integrity and clean up any resources that are no longer needed that were allocated to perform the connection that subsequently failed. But to maintain correct layering the engine shouldn't know whether the connection was attempted as a result of an end user request or because some background task was being performed. In the latter case, notifying the end user would be confusing since they'd have no idea what the error meant as they wouldn't have initiated the connection attempt.
- The engine cannot report errors to the end-user because not only does it not know if this is appropriate but doing so would violate the layering of the application and make future maintenance more difficult.
- Ignoring errors is also not an option since this might be an important operation which the user expects to run reliably. Ignoring an error

¹⁵This is especially true for system-critical threads since any panic in such threads will cause the entire device to reboot.

¹⁶For more information see [Campbell *et al.*, 2007, Chapter 3].

might even cause the user's data to be corrupted¹⁷ so it is important the whole application is designed to use the most appropriate error-handling strategy to resolve any problems.

For system errors, such as `KErrNoMemory` resulting from the failure to allocate a resource, you might think that a valid approach to resolving this error would be to try to free up any unnecessary memory. This would resolve the error within the engine with no need to involve the application at all. But how would you choose which memory to free? Clearly all memory has been allocated for a reason and most of it to allow client requests to be serviced. Perhaps caches and the like can be reduced in size but that will cause operations to take longer. This might be unacceptable if the application or service has real-time constraints that need to be met.

Solution

Lower-level components should not try to handle domain or system errors silently unless they have the full context and can do so successfully with no unexpected impact on the layers above. Instead lower layers should detect errors and pass them upwards to the layer that is capable of correctly resolving them.

Symbian OS provides direct support for escalating errors upwards known as the *leave* and *trap* operations. These allow errors to be propagated up the call stack by a *leave* and trapped by the layer that has sufficient context to resolve it. This mechanism is directly analogous to exception handling in C++ and Java.

Symbian OS does not explicitly use the standard C++ exception-handling mechanism, for historical reasons. When Symbian OS, or EPOC32 as it was then known, was first established, the compilers available at that time had poor or non-existent support for exceptions. You can use C++ exceptions within code based on Symbian OS. However, there are a number of difficulties with mixing *Leave* and *trap* operations with C++ exceptions and so it is not recommended that you use C++ exceptions.

Structure

The most basic structure for this pattern (see Figure 2.3) revolves around the following two concepts:

- The *caller* is the higher-layer component that makes a function call to another component. This component is aware of the motivation for the function call and hence how to resolve any system or domain errors that might occur.

¹⁷One of the cardinal sins on Symbian OS.

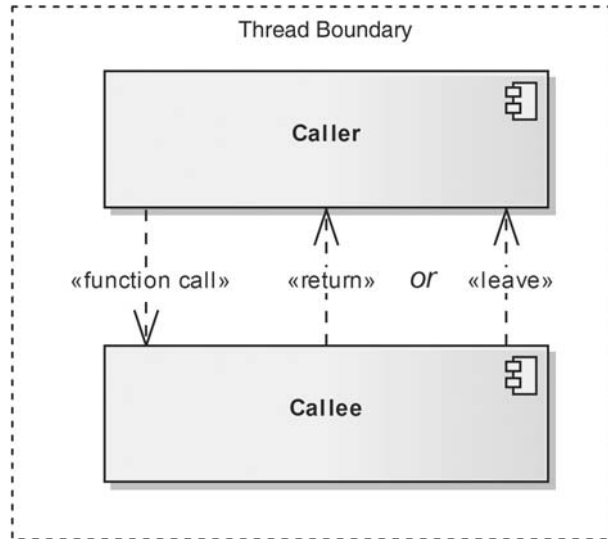


Figure 2.3 Structure of *Escalate Errors* pattern

- The *callee* is the lower-layer component on which the function call is made. This component is responsible for attempting to satisfy the function call if possible and detecting any system or domain errors that occur whilst doing so. If an error is detected then the function escalates this to the caller through the Leave operation, otherwise the function returns as normal.

An important point to remember when creating a function is that there should only be one path out of it. Either return an error or leave but do not do both as this forces the caller to separately handle all of the ways an error can be reported. This results in more complex code in the caller and usually combines the disadvantages of both approaches!

Note that all leaves must have a corresponding trap harness. This is to ensure that errors are appropriately handled in all situations.¹⁸ A common strategy for resolving errors is to simply report the problem, in a top-level trap harness, to the end user with a simple message corresponding to the error code.

Dynamics

Normally, a whole series of caller–callee pairs are chained together in a call stack. In such a situation, when a leave occurs, the call stack is

¹⁸If a Leave doesn't have a trap associated with it, your thread will terminate with a USER 175 panic since there's little else Symbian OS can do.

unwound until control is returned to the closest trap. This allows an error to be easily escalated upwards through more than one component since any component that doesn't want to handle the error simply doesn't trap it (see Figure 2.4).

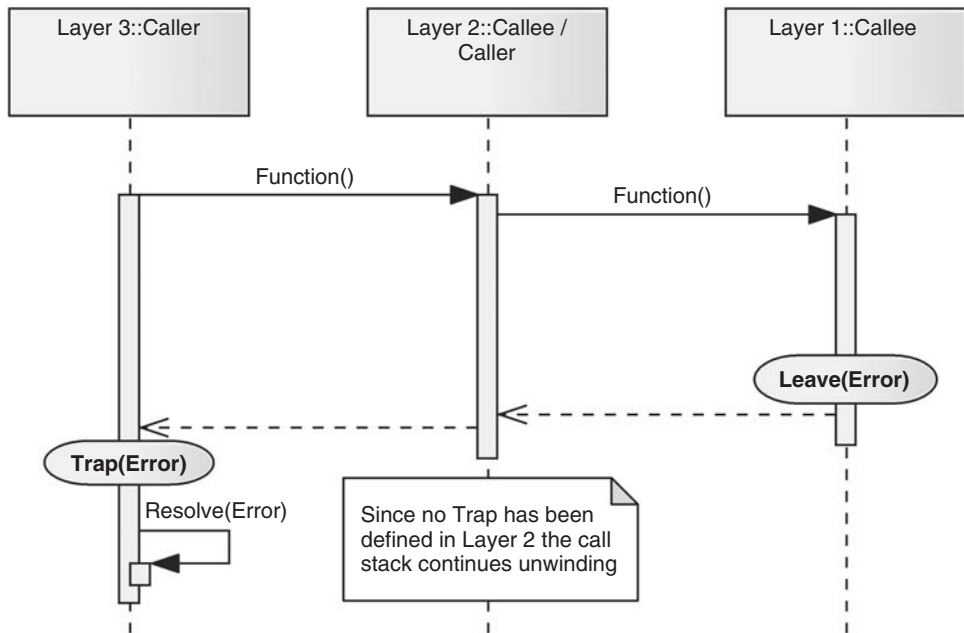


Figure 2.4 Dynamics of *Escalate Errors* pattern

The most important decision to make is where you should place your trap harnesses. Having coarse-grained units of recovery has the advantage of fewer trap harnesses and their associated recovery code but with the disadvantage that the recovery code may be general and complex. There is also the danger that a small error leads to catastrophic results for the end user. For instance, if not having enough memory to apply bold formatting in a word processor resulted in a leave that unwound the entire call stack, this might terminate the application without giving the end user the opportunity to save and hence they might lose their data! On the other hand, too fine-grained units of recovery results in many trap harnesses and lots of recovery code with individual attention required to deal with each error case as well as a potentially significant increase in the size of your executable.

Unlike other operating systems, Symbian OS is largely event-driven so the current call stack often just handles a tiny event, such as a keystroke or a byte received. Thus trying to handle every entry point with a separate

trap is impractical. Instead leaves are typically handled in one of three places:

- Many threads have a top-level trap which is used as a last resort to resolve errors to minimize unnecessary error handling in the component. In particular, the Symbian OS application framework provides such a top-level trap for applications. If a leave does occur in an application, the `CEikAppUi::HandleError()` virtual function is called allowing applications to provide their own error-handling implementation.
- Traps are placed in a `RunL()` implementation when using *Active Objects* (see page 133) to handle the result of an asynchronous service call. Or alternatively you can handle the error in the corresponding `RunError()` function if your `RunL()` leaves.
- Trap harnesses can be nested so you do not need to rely on just having a top-level trap. This allows independent sub-components to do their own error handling if necessary. You should consider inserting a trap at the boundary of a component or layer. This can be useful if you wish to attempt to resolve any domain errors specific to your component or layer before they pass out of your control.

Implementation

Leaves

A leave is triggered by calling one of the `User` leave functions defined in `e32std.h` and exported by `euser.dll`. By calling one of these functions you indicate to Symbian OS that you cannot finish the current operation you are performing or return normally because an error has occurred. In response, Symbian OS searches up through the call stack looking for a trap harness to handle the leave. Whilst doing so Symbian OS automatically cleans up objects pushed onto the cleanup stack by lower-level functions.

The main leave function is `User::Leave(TInt aErr)` where the single integer parameter indicates the type of error and is equivalent to a throw statement in C++. By convention, negative integers are used to represent errors. There are a few helper functions that can be used in place of `User::Leave()`:

- `User::LeaveIfError(TInt aReason)` leaves if the reason code is negative or returns the reason if it is zero or positive.
- `User::LeaveIfNull(TAny* aPtr)` leaves with `KErrNoMemory` if `aPtr` is null.

- `new(ELeave) CObject()` is an overload of the `new` operator that automatically leaves with `KErrNoMemory` if there is not enough memory to allocate the object on the heap.

Here is an example where a function leaves if it couldn't establish a connection to the file server due to some system error or because it couldn't find the expected file. In each case, the higher layers are given the opportunity to resolve the error:

```
void CEngine::ConstructL(const TDesC& aFileName)
{
    RFs fs;
    User::LeaveIfError(fs.Connect());

    RFile file;
    User::LeaveIfError(file.Open(fs, aFileName, EFileRead));
    ...
}
```

By convention, the names of functions which can leave should always be suffixed with an 'L' (e.g., `ConstructL()`) so that a caller is aware the function may not return normally. Such a function is frequently referred to as a *leaving function*. Note that this rule applies to any function which calls a leaving function even if does not call `User::Leave()` itself. The function implicitly has the potential to leave because un-trapped leaves are propagated upward from any functions it calls.

Unfortunately you need to remember that this is only a convention and is not enforced by the compiler so an 'L' function is not always equivalent to a leaving function. However, static analysis tools such as `epoc32\tools\leavescan.exe` exist to help you with this. These tools parse your source code to evaluate your use of trap and leave operations and can tell you if you're violating the convention. They also check that all leaves have a trap associated with them to help you avoid USER 175 panics.

Traps

A trap harness is declared by using one of the `TRAP` macros defined in `e32cmn.h`. These macros will catch any leave from any function invoked within a `TRAP` macro. The main trap macro is `TRAP(ret, expression)` where `expression` is a call to a leaving function¹⁹ and `ret` is a pre-existing `TInt` variable. If a leave reaches the trap then the operating system assigns the error code to `ret`; if the expression returns normally, without leaving or because the leave was trapped at a lower

¹⁹To be precise it doesn't have to be a leaving function but then, if it isn't, why would you trap it?

level in the call stack, then `ret` is set to `KErrNone` to indicate that no error occurred.

As the caller of a function within a trap you should not need to worry about resources allocated by the callee. This is because the leaving mechanism is integrated with the Symbian OS cleanup stack. Any objects allocated by the callee and pushed onto the cleanup stack are deleted prior to the operating system invoking the trap harness.

In addition to the basic `TRAP` macro, Symbian OS defines the following similar macros:

- `TRAPD(ret, expression)` – the same as `TRAP` except that it automatically declares `ret` as a `TInt` variable on the stack for you (hence the ‘D’ suffix) for convenience.
- `TRAP_IGNORE(expression)` – simply traps `expression` and ignores whether or not any errors occurred.

Here is an example of using a trap macro:

```
void CMyComponent::Draw()
{
    TRAPD(err, iMyClass->AllocBufferL());
    if(err < KErrNone)
    {
        DisplayErrorMsg(err);
        User::Exit(err);
    }
    ... // Continue as normal
}
```

Intermediate Traps

If a function traps a leave but then determines from the error code that it is unable to resolve that specific error, it needs to escalate the error further upwards. This can be achieved by calling `User::Leave()` again with the same error code.

```
TRAPD(err, iBuffer = iMyClass->AllocBufferL());
if(err < KErrNone)
{
    if(err == KErrNoMemory)
    {
        // Resolve error
    }
    else
    {
        User::Leave(err); // Escalate the error further up the call stack
    }
}
```


Trapping and leaving again is normally only done if a function is only capable of resolving a subset of possible errors and wishes to trap some while escalating others. This should be done sparingly since every intermediate trap increases the cost of the entire leave operation as the stack unwind has to be restarted.

Additional Restrictions on Using Trap–Leave Operations

- You should not call a leaving function from within a constructor. This is because any member objects that have been constructed will not have their destructors called which can cause resource leaks. Instead you should leave from within a `ConstructL()` method called during the standard Symbian OS two-stage construction process as described by [Babin, 2007, Chapter 4].
- You also should not allow a Leave to escape from a destructor. Essentially this means that it is permissible to call leaving functions within a destructor so long as they are trapped before the destructor completes. This is for two reasons; the first is that the leave and trap mechanisms are implemented in terms of C++ exceptions and hence if an exception occurs the call stack is unwound. In doing so the destructors are called for objects that have been placed on the call stack. If the destructors of these objects leave then an abort may occur on some platforms as Symbian OS does not support leaves occurring whilst a leave is already being handled.²⁰
The second reason is that, in principle, a destructor should never fail. If a destructor can leave, it suggests that the code has been poorly architected. It also implies that part of the destruction process might fail, potentially leading to memory or handle leaks. One approach to solving this is to introduce ‘two-phase destruction’ where some form of `ShutdownL()` function is called prior to deleting the object. For further information on this, see the Symbian Developer Library.

Consequences

Positives

- Errors can be handled in a more appropriate manner in the layer that understands the error compared to attempting to resolve the error immediately.
Escalating an error to a design layer with sufficient context to handle it ensures that the error is handled correctly. If this is not done and an attempt is made to handle an error at too low a level, your options for

²⁰Also known as a nested exception.

handling the error are narrowed to a few possibilities which are likely to be unsuitable.

The low-level code could retry the failed operation; it could silently ignore the error (not normally practical but there may be circumstances when ignoring certain errors is harmless); or it could use *Fail Fast* (see page 17). None of these strategies is particularly desirable especially the use of *Fail Fast*, which should be reserved for faults rather than the domain or system errors that we are dealing with here.

In order to handle an error correctly without escalating it, the component would probably be forced to commit layering violations, e.g., by calling up into the user interface from lower-level code. This mixing of GUI and service code causes problems with encapsulation and portability as well as decreasing your component's maintainability. This pattern neatly avoids all these issues.

- Less error-handling code needs to be written, which means the development costs and code size are reduced as well as making the component more maintainable.

When using this pattern, you do not need to write explicit code to check return codes because the leave and trap mechanism takes care of the process of escalating the error and finding a function higher in the call stack which can handle it for you. You do not need to write code to free resources allocated by the function if an error occurs because this is done automatically by the cleanup stack prior to the trap harness being invoked. This is especially true if you use a single trap harness at the top of a call stack which is handling an event.

- Runtime performance may be improved.
Use of leave-trap does not require any logic to be written to check for errors except where trap harnesses are located. Functions which call leaving functions but do not handle the leaves themselves do not have to explicitly propagate errors upwards. This means that efficiency during normal operation improves because there is no need to check return values to see if a function call failed or to perform manual cleanup.

Negatives

- Traps and leaves are not as flexible as the C++ exception mechanism. A leave can only escalate a single `TInt` value and hence can only convey error values without any additional context information. In addition, a trap harness cannot be used to catch selected error values. If this is what you need to do then you have to trap all errors and leave again for those that you can't resolve at that point which is additional code and a performance overhead for your component.

- Runtime performance may get worse when handling errors. A leave is more expensive in terms of CPU usage, compared to returning an error code from a function, due to the cost of the additional machinery required to manage the data structures associated with traps and leaves. In the Symbian OS v9 Application Binary Interface (ABI), this overhead is currently minimal because the C++ compiler's exception-handling mechanism is used to implement the leave-trap mechanism which is usually very efficient in modern compilers.

It is best to use leaves to escalate errors which are not expected to occur many times a second. Out-of-memory and disk-full errors are a good example of non-fatal errors which are relatively infrequent but need to be reported and where a leave is usually the most effective mechanism. Frequent leaves can become a very noticeable performance bottleneck. Leaves also do not work well as a general reporting mechanism for conditions which are not errors. For example, it would not be appropriate to leave from a function that checks for the presence of a multimedia codec capability when that capability is not present. This is inefficient and leads to code bloat due to the requirement on the caller to add a trap to get the result of the check.

- Leaves should not be used in real-time code because the leave implementation does not make any real-time guarantees due to the fact that it involves cleaning up any items on the cleanup stack and freeing resources, usually an unbounded operation.
- Without additional support, leaves can only be used to escalate errors within the call stack of a single thread.
- This pattern cannot be used when writing code that forms part of the Symbian OS kernel, such as device drivers, because the leave-trap operations are not available within the kernel.

Example Resolved

In the example, an application wished to send data to a peer via UDP. To do this, it was divided into two layers: the UI, dealing with the end user, and the engine, dealing with Comms-Infras.

Engine Layer

To achieve this, we need to open a UDP connection to be able to communicate with the peer device. The `RSocket::Open()` function opens a socket and `RSocket::Connect()` establishes the connection. These operations will fail if Comms-Infras has insufficient resources or the network is unavailable. The engine cannot resolve these errors because it is located at the bottom layer of the application design and does not

have the context to try to transparently recover from an error without potentially adversely affecting the end user. In addition, it has no way of releasing resources to resolve local resource contention errors because they are owned and used by other parts of the application it does not have access to.

We could implement escalation of the errors by using function return codes²¹ as follows:

```
TInt CEngine::SendData(const TDesC8& aData)
{
    // Open the socket server and create a socket
    RSocketServ serv;
    TInt err = serv.Connect();
    if(err < KErrNone)
    {
        return err;
    }

    RSocket sock;
    err = socket.Open(serv,
                     KAfinet,
                     KSockDatagram,
                     KProtocolInetUdp);
    if(err < KErrNone)
    {
        serv.Close();
        return err;
    }

    // Connect to the localhost.
    TInetAddr addr;
    addr.Input(_L("localhost"));
    addr.SetPort(KTelnetPort);
    TRequestStatus status;
    sock.Connect(addr, status);
    User::WaitForRequest(status);
    if(status.Int() < KErrNone)
    {
        sock.Close();
        serv.Close();
        return status.Int();
    }

    // Send the data in a UDP packet.
    sock.Send(aData, 0, status);
    User::WaitForRequest(status);

    sock.Close();
    serv.Close();
    return status.Int();
}
```

²¹Note that the code is given to show how the RSocket API handles errors and is not production quality. For instance, a normal application would use *Active Objects* (see page 133) instead of TRequestStatus objects.

However, as you can see, the error-handling code is all mixed up with the normal flow of execution making it more difficult to maintain. A better approach would be to use the Symbian OS error-handling facilities, resulting in a much more compact implementation:

```
void CEngine::SendDataL(const TDesC8& aData)
{
    // Open the socket server and create a socket
    RSocketServ serv;
    User::LeaveIfError(serv.Connect());
    CleanupClosePushL(serv);

    RSocket sock;
    User::LeaveIfError(sock.Open(serv,
                                KAfInet,
                                KSockDatagram,
                                KProtocolInetUdp));

    CleanupClosePushL(sock);

    // Connect to the localhost.
    TInetAddr addr;
    addr.Input(_L("localhost"));
    addr.SetPort(KTelnetPort);
    TRequestStatus status;
    sock.Connect(addr, status);
    User::WaitForRequest(status);
    User::LeaveIfError(status.Int());

    // Send the data in a UDP packet.
    sock.Send(aData, 0, status);
    User::WaitForRequest(status);
    User::LeaveIfError(status.Int());

    CleanupStack::PopAndDestroy(2); // sock and serv
}
```

Note that in the above we rely on the fact that `RSocket::Close()` does not leave. This is because we use `CleanupClosePushL()` to tell the cleanup stack to call `Close()` on both the `RSocketServ` and `RSocket` objects if a leave occurs while they're on the cleanup stack. This is a common property of Symbian OS functions used for cleanup functions, such as `Close()`, `Release()` and `Stop()`. There is nothing useful that the caller can do if one of these functions fails, so errors need to be handled silently by them.

UI Layer

In this case, the application implementation relies on the application framework to provide the top-level trap harness to catch all errors escalated upwards by the Engine. When an error is caught by the trap it then calls the `CEikAppUi::HandleError()` virtual function. By default, this displays the error that occurred in an alert window to the end user. If

you've put everything on the cleanup stack then this may be all you need to do. However, an alternative is to override the function and provide a different implementation. Note that `HandleError()` is called with a number of parameters in addition to the basic error:

```
TErrorHandlerResponse CEikAppUi::HandleError(TInt aError,
                                             const SExtendedError& aExtErr,
                                             TDes& aErrorText,
                                             TDes& aContextText)
```

These parameters are filled in by the application framework and go some way to providing extra context that might be needed when resolving the error at the top of the application's call stack. By relying on this, the lower layers of the application can escalate any errors upwards to the top layer in the design to handle the error. Use of this pattern enables errors to be resolved appropriately and minimizes the amount of error-handling code which needs to be written.

Other Known Uses

This pattern is used extensively within Symbian OS so here are just a couple of examples:

- *RArray*
This is just one of many classes exported from `euser.dll` that leaves when it encounters an error. Basic data structures like these don't have any knowledge of why they're being used so they can't resolve any errors. Interestingly, this class provides both a leave and a function return variant of each of its functions. This is so that it can be used kernel-side, where leaves cannot be used, and user-side, where leaves should be used to simplify the calling code as much as possible.
- *CommsDat*
`CommsDat` is a database engine for communication settings such as network and bearer information. It is rare for an error to occur when accessing a `CommsDat` record unless the record is missing. Hence by using leaves to report errors, its clients can avoid having to write excessive amounts of error-handling code. The use of the leave mechanism also frees up the return values of functions in the APIs so that they can be used for passing actual data.

Variants and Extensions

- *Escalating Errors over a Process Boundary*
A limitation of the basic trap and leave operations is that they just escalate errors within a single thread. However, the Symbian OS

Client–Server framework extends the basic mechanism so that if a leave occurs on the server side it is trapped within the framework. The error code is then used to complete the IPC message which initiated the operation that failed. When received on the client side this may be converted into a leave and continue being escalated up the client call stack. However, this is dependent on the implementation of the client-side DLL for the server.

- *Escalating Errors without Leaving*
Whilst this pattern is normally implemented using the leave and trap mechanisms this isn't an essential part of the pattern. In situations where the higher layer didn't originate the current call stack, it may not be possible to use a Leave to escalate the error back to it. Instead the error will need to be passed upwards via an explicit function call which informs the higher layer of the error that needs to be handled. Examples of this commonly occur when an asynchronous request fails for some reason since by the time the response comes back the original call stack no longer exists; for instance, this occurs in *Coordinator* (see page 211) and *Episodes* (see page 289).

References

- *Fail Fast* (see page 17) is an alternative pattern to this pattern that is intended to solve the problem of handling faults. In general, the two patterns directly conflict and should not be used to handle the same errors. They are designed for different situations.
- *Client–Server* (see page 182) extends the basic trap and leave operations to escalate errors from the server side into the client thread.
- The SDL provides further information on using the cleanup stack on Symbian OS at Symbian Developer Library » Symbian OS guide » Base » Using User Library (E32) » Memory Management » Using Cleanup Support.
- [Longshaw and Woods, 2004] discusses similar problems in a different context.

3

Resource Lifetimes

This chapter contains a collection of patterns about when and how to allocate resources, but before we go any further we need to understand what we mean by a resource. For our purposes, a resource is defined as any physical or virtual entity of limited availability that needs to be shared across the system. This can be broken down into two further types:

- *Ephemeral resources*, such as CPU cycles or battery power, have one key trait: you can't hold on to them or reserve them for your own purposes. They're there but you can't keep a grip on them whether you use them or not.
- *Ownable resources*, such as RAM, disk space and communication channels, are resources that you can hold onto and keep using. Such resources must normally be reserved for your sole use. During the period that they are allocated to you, these resources cannot be shared or borrowed by anyone else until you de-allocate them.

In this chapter, we'll be focusing on ownable resources and how to balance the non-functional characteristics of sharing them across a system.

Whatever their type, the resources available to software on a mobile device, are much more constrained than on your average laptop or desktop PC. Compared to mainstream laptops, mobile devices are between 10 and 30 times smaller in terms of CPU speed, RAM size and storage space and are set to remain so in relative terms even as they increase in absolute terms. This is partly to do with the fact that mobile devices are expected to be carried around in your pocket all the time putting both size and weight constraints on the devices. Another factor is that,

with over 7 million Symbian OS devices now being sold each month,¹ device manufacturers are keen to maximize their profits by keeping the cost of the materials in each device as low as possible. The reason for this is that to add each additional resource incurs additional hardware costs for device manufacturers. For instance, increasing the instructions processed per second means including a faster CPU whilst boosting the memory available to the system means including a bigger RAM chip. When shipping potentially millions of each product, even small savings per device add up quickly. Also, if an end user finds a device frequently running out of resources they can't just plug in a new bit of hardware as they could on a PC. The result of this is that, as a developer of software based on Symbian OS, you have to carefully design your code to take account of the scarcity of resources. In general, this means considering how to minimize your use of each type of resource and building in the expectation that at some point a resource you rely on is going to run out.

The problem is that you can go too far. If every resource were to be allocated, used and de-allocated immediately, the resources used across the system would certainly be minimized. However, end users probably wouldn't care because the device would be so slow as to be unusable. Continually allocating and de-allocating all ownable resources would be non-trivial and would take a significant amount of time and CPU cycles. If you've ever profiled an application, you've probably seen the time spent allocating memory in functions taking a significant proportion of the total time. This is in addition to the introduction of many more error paths which would increase the complexity and fragility of your software. However, reserving a resource for your own use means that the next request for a similar type of resource elsewhere in the system either increases the total resource usage in the system or fails.

A key decision you need to make is when and how much you should allocate and de-allocate resources. You'll need to consider the following factors:

- Software throughout a device needs to co-operate in using resources and avoid holding onto them unnecessarily.
- Software startup needs to be quick, whether it is for the device, an application or a service.
- The software needs to respond to an event, such as the user pressing a button, with the minimum of delay.
- Some resource allocations are potentially unbounded operations and hence must be avoided in real-time use cases. One example

¹ www.symbian.com/news/pr/2008/pr20089781.html.

of this is allocating memory using the default Symbian OS allocator.

- Allocating resources may fail and so may introduce additional error paths that must be dealt with. For instance, a resource might need to always be available so that a particular action cannot fail.
- Improving a base allocation mechanism² is going to be hard. This is because they've usually been optimized already. For instance, [Berger *et al.*, 2002] showed that six out of eight applications using custom memory allocators performed better when using a state-of-the-art, general-purpose allocator. Also, if you don't have access to the source code for the base allocation mechanism then you don't even have the opportunity to improve on it without starting again from scratch. When allocating resources other than memory, there could also be unavoidable delays such as network latency.

These forces are often contradictory and have to be traded off against one another based on the following kinds of factors:

- where in the execution path the resource needs to be used
- how frequently the resource will be used
- who uses the resource
- the consequences to the end user's experience due to the knock-on impact of failing to allocate a resource.

A way of balancing these forces is to use the above factors to decide what the lifetime of your use of a resource should be. You need to choose somewhere between the two extremes:

- *Mayfly*³ – you allocate an object, use it and de-allocate it. This is the default lifetime and gives the absolute minimum use of the resource at a potentially high cost in terms of execution time and additional error paths.
- *Immortal* – you allocate a resource as soon as possible and never de-allocate it. This is the opposite of Mayfly and gives the best performance and reliability at the expense of the highest resource usage.

The first pattern in this series, *Immortal* (see page 53), explains the effects of choosing one of these extremes. The subsequent patterns, *Lazy*

²By this I mean going from the state of the resource having no users at all to it being used.

³An insect well known for its very short lifetime (en.wikipedia.org/wiki/Mayfly). Also known as Variable Allocation in [Weir and Noble, 2000].

Allocation (see page 63) and *Lazy De-allocation* (see page 73), discuss how a compromise between the two can be found.

Each of these patterns is described in terms of the following main objects:

- A *resource* represents an ownable resource of a specific type.
- A *resource provider* provides an interface through which a resource object can be allocated or de-allocated. When a resource has been allocated, it will not be provided to any other client until it has been de-allocated by the original client.
- A *resource client* uses the resource provider interface to reserve one or more resource objects by allocating them for its own use. The resource client then has the responsibility for deciding how long it wishes to hold on to the resource objects before handing them back the resource provider by de-allocating them.

These patterns focus on what resource clients can do to affect the lifetime of a resource. This is because a common problem is that resource clients will override strategies employed by resource providers to reduce resource usage by holding onto a resource unnecessarily. However, for each of the patterns discussed in this chapter you could create a variant in which the resource provider takes more of the responsibility for managing resource lifetimes.

Immortal

Intent Allocate a resource as soon as possible so that, when you come to use it later, it is guaranteed to be both available and quickly accessible.

AKA Static Allocation, Pre-allocation, Fixed Allocation [Weir and Noble, 2000]

Problem

Context

A resource is required and the details of how it will be used are well understood. There is a critical need for its allocation, as well as the possible failure of the allocation, to be controlled.

Summary

You need to resolve one or more of the following:

- You would like to guarantee that allocating the resource will succeed.
- The responsiveness of your application or service is negatively impacted by the time to allocate one or more resources.
- You need to avoid any unbounded resource allocations during a real-time use case.

Whichever of the above applies in your situation, it must also be more important that your application or service always has access to the resource than it is to share the resource with other potential resource clients.

Description

On most mobile devices, some applications or services are considered essential. What is essential may be mandated by the mobile device creator (e.g. a large-mega-pixel camera, if it is an essential selling point for a specific device) or by regulatory bodies (e.g. the ability to make a call to the emergency services). In all normal situations, an essential service should not fail.⁴ Hence these services should ensure that all the possible resources that they need are guaranteed to be available. However, in an

⁴Of course in abnormal situations, for example if the device's battery is dying, it might be reasonable that the essential service does not work.

open system such as Symbian OS, resources cannot be guaranteed to be available if the allocation is done on demand. For instance, it might be that there is not enough memory to allocate all the required objects or that a communication port is in use by someone else.

Alternatively, your problem could be that your software wouldn't be very responsive if you took the time to allocate all the resources needed to handle an event. For instance, you might need to have an open network connection through which to send data before you can tell the user the operation was successful. Having to open the network connection just to send the data each time could ruin the experience for the end user if done repeatedly.

Lastly, any real-time use case cannot be performed if a resource allocation within it either takes an unbounded amount of time or simply takes so long that the required deadlines cannot be met. One example of such a resource is any memory allocated using the default Symbian OS allocator.

Any of these problems might be experienced by any type of Symbian OS developer.

Example

In many parts of the world, there is a trend to towards ever more accurate location information being automatically provided to the emergency services when a user makes an emergency phone call on their mobile phone (see, for example, the EU's Universal Service Directive [EU, 2002] and the USA's Enhanced 911 – Wireless Services specification [FCC, 2001]). To fulfill these, the Symbian OS Location-Based Services (LBS) subsystem must always be able to provide a location for an emergency network location request. It would not be acceptable to fail to do this as the result of unsuccessfully allocating a resource in the process of handling the request.

Solution

Allocate all the immortal resources needed when your application or service starts but before they are actually needed. Any failure to allocate an immortal resource should result in the entire application or service failing to start.

Structure

Both the resource and the resource provider are as described in the introduction to this chapter. However, the resource client is also responsible for pre-allocating the immortal resource as soon as it is created (see Figure 3.1).

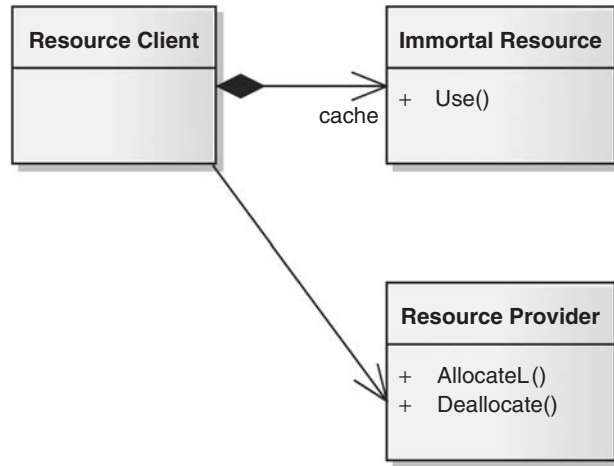


Figure 3.1 Structure of the *Immortal* pattern

Dynamics

The following phases of activity occur at run time (see Figure 3.2):

- *Construction Phase* – when the immortal resource is allocated and held until it is needed. Any failure at this stage results in the resource client object failing to be constructed (see also *Fail Fast* on page 17). This phase always happens once, for a particular resource, and occurs as soon as the application or service is started to try to avoid contention with other potential resource clients.
- *Usage Phase* – when the immortal resource has already been allocated and can be used without any further initialization. This phase may occur multiple times after the Construction Phase.
- *Destruction Phase* – The resource client only de-allocates the immortal resource when the resource client is itself destroyed.

Implementation

The main thing to decide when using this pattern in your design is what constitutes an immortal resource and hence needs to be pre-allocated. It should at least meet the following criteria:

- It is a resource without which your application can not function.
- You are able to fully specify the immortal resources needed during development. For instance, you know the total size of RAM or which communication channels are needed and appreciate the impact on the system due to not sharing these resources.

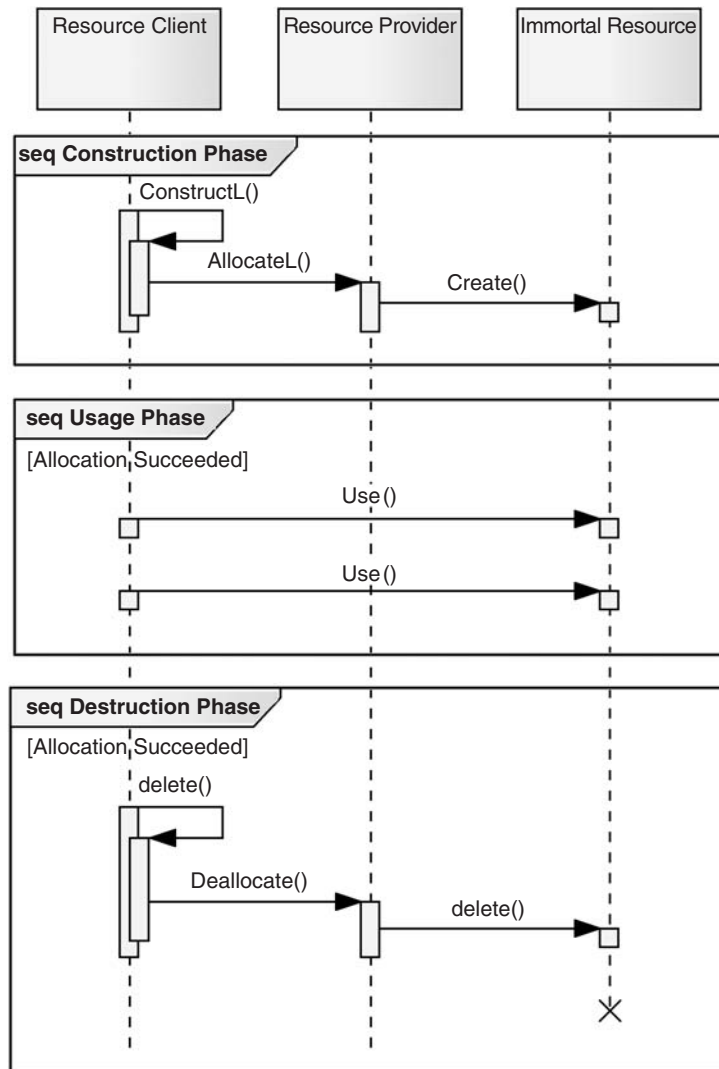


Figure 3.2 Dynamics of the *Immortal* pattern

Having identified the resource, there are a number of ways of implementing the allocation operation depending on the exact type of your resource. Here are some of the more common methods.

Pre-allocation via a C Class

In this case, one object provides both the resource and the resource provider interfaces:


```

class CResource : public CBase
{
public:
    static CResource* NewL(); // Allocator
    ~CResource();           // De-allocator

    void Use();
};

```

```

class CResourceOwner : public CBase
{
public:
    static CResourceOwner* NewL();

    void DoStuff();

private:
    void ConstructL();

private:
    // When you're pre-allocating a single object:
    CResource* iPreAllocatedResource;

    // When you're pre-allocating a number of objects:
    TFixedArray<CResource*, KNumPreAllocatedResources> iPreAllocatedArray;
};

```

```

// All allocation error paths are dealt with upfront during the
// construction of the resource client. If a resource is unavailable
// then a Leave will occur.
void CResourceOwner::ConstructL()
{
    // Pre-allocation of a single object
    iPreAllocatedResource = CResource::NewL();

    // Pre-allocation of multiple objects
    for (TInt index = 0; index < KNumPreAllocatedResources; index++)
    {
        iPreAllocatedArray[index] = CResource::NewL();
    }
}

// Note the lack of error paths
void CResourceOwner::DoStuff()
{
    iPreAllocatedResource->Use();

    for (TInt index = 0; index < KNumPreAllocatedResources; index++)
    {
        iPreAllocatedArray[index]->Use();
    }
}

```

```
CResourceOwner::~CResourceOwner()
{
    delete iPreAllocatedResource;
    iPreAllocatedArray.DeleteAll();
}
```

For more information on the `TFixedArray` class, see the Symbian Developer Library.

In some cases your resource will contain some state within it which you don't want to be persistent across the whole lifetime of the resource client.⁵ To avoid this problem you may need to reset the resource object before using it in the `DoStuff()` function. If that is the case then you should design your resource to have an initialization function separate from the constructor. However, you should be careful to ensure that this function doesn't leave or you'll have introduced an error path which is exactly what you don't want.

Pre-allocation via an R Class

In this case, the resource provider acts as a Proxy [Gamma *et al.*, 1994] for the actual resource. Hence the resource client doesn't actually have a pointer directly to the resource itself but accesses it through the resource provider:

```
class RResourceProvider
{
public:
    TInt OpenL(); // Allocator
    void Close(); // De-allocator
    void Use();

private:
    CResource* iResource; // Owned by another class
};
```

```
class CResourceOwner : public CBase
{
public:
    static CResourceOwner* NewL(); // Calls ConstructL()

    void DoStuff();

private:
    void ConstructL();

private:
    RResourceProvider iPreAllocatedResource;
};
```

⁵Such as the contents of a buffer that isn't valid between different calls to `CResourceOwner::DoStuff()`.

```
// All allocation error paths are dealt with upfront during the
// construction of the resource client. If a resource is unavailable
// then a Leave will occur.
void CResourceOwner::ConstructL()
{
    iPreAllocatedResource.OpenL();
}

// Note the lack of error paths
void CResourceOwner::DoStuff()
{
    iPreAllocatedResource.Use();
}

CResourceOwner::~CResourceOwner()
{
    iPreAllocatedResource.Close();
}
```

In this case, `RResourceProvider` is actually a handle to the real resource somewhere else. For instance, if `RResourceProvider` were in fact an `RSocket` being used to hold open a network connection then `CResourceOwner` wouldn't own any additional memory. It would however be responsible for causing the socket server to use more resources – mainly the communication port but also memory for supporting objects as well as ephemeral resources such as power for the radio antenna to broadcast 'I'm still here' signals to the network and CPU cycles to maintain the port in an open state.

Consequences

Positives

- You guarantee the resource is available at any point after the Construction Phase. This is often necessary to support critical use cases where any error occurring due to a failed allocation would be completely unacceptable.
- You control when and how a resource is allocated irrespective of how it is used.
- Immortal resources respond more quickly when used since they do not need to be allocated each time.
- Using immortal resources reduces the churn in the total set of that resource type, which can reduce fragmentation.
- There is a reduced chance of leaking a resource and so wasting the system resources over time since resource de-allocations occur less frequently.
- Your component is easier to test since its use of the resource is predictable.

Negatives

- Pre-allocating a resource means that it cannot be used by any other process in the system. This can be acceptable in closed systems but, on an open platform such as Symbian OS, this often causes contention problems between applications and services, especially if there aren't many instances of the immortal resource. Even if the size of the immortal resource is small compared to the total amount of the resource, there may still be problems if this pattern is applied across the system many times.
- Usability of the device may be reduced because, whilst one component is using a resource, another may not be able to perform its function until that resource has been de-allocated and is free for general use again. This reduces the number of use cases that a device can perform concurrently, which is particularly a problem on high-end devices.
- This pattern can result in waste since the resource usage reflects the worst-case scenario. This is particularly a problem if there are long periods between each use of the resource or if the resource is not actually used during the lifetime of the owning application or service.
- Starting applications or services with immortal resources takes longer.
- The use of this pattern prevents other lifecycles being used 'under the hood' for individual resources by resource providers on behalf of resource clients.

Example Resolved

As already mentioned, the LBS subsystem needs to be able to guarantee to satisfy an emergency network location request. This is true even if multiple applications are using the subsystem or if there is no more memory available in the system: a location must still be provided. Only if there is no more power available for the device should the subsystem fail to provide a location.⁶

To achieve this, the following strategies are employed by the LBS subsystem:

- Mobile devices that support emergency network location requests must choose the Full Mode configuration of the LBS subsystem. This ensures that, when the device is started, the LBS subsystem is included in the boot-up sequence. Hence all five of the LBS processes are pre-allocated at a default cost of at least 105–170 KB of RAM.⁷

⁶Power is an ephemeral resource and cannot be pre-allocated for one particular use.

⁷See Appendix A for more details.

However, more than just memory has been reserved as this approach also ensures that any dynamic configuration data is available and that there's no subsequent possibility of a corrupt INI file generating additional errors at just the wrong moment.

- Messages between components within the LBS subsystem are sent via *Publish and Subscribe* (see page 114). This is because, when the subsystem is created, each of its components are able to pre-allocate all of the resources it needs, by calling `RProperty::Define()`, to maintain its IPC channels even if the system is out of memory.

For instance, this means that the Network Request Handler, which processes location requests from remote parties, can guarantee that it is able to send commands directly to the A-GPS module. Similarly, once the A-GPS module has obtained the position data from the GPS hardware it can guarantee to publish the location data and have it received by the location server.

- Only the immortal resources required to generate a location are pre-allocated. For instance, under normal circumstances the subsystem can be configured so that the user is notified before a location is sent over the network. However, in out-of-memory situations this notification to the user is not guaranteed since it was decided that the complexity involved in arranging for sufficient immortal resources to do the job wasn't necessary. This is because an end user is unlikely to complain if they weren't informed that the emergency services were told where to rescue them from!⁸

Other Known Uses

- *Camera Hardware*

Some types of hardware usually require resources to be allocated by their device driver when a device is booted. For instance, large-mega-pixel cameras require correspondingly large contiguous memory buffers so the device driver always has sufficient memory to store an image. This memory is a significant proportion of the entire memory of the device and needs to be contiguous for performance reasons. If cameras didn't pre-allocate the memory, they would run a high risk of failing to allocate the memory on demand due to fragmentation of the device's RAM, even if the total memory available would be enough to satisfy the request.⁹

Interestingly, camera applications normally do not choose to reserve sufficient disk space to store a picture. This is because the disk space required is normally a smaller proportion of the total available but

⁸If it is not an emergency then the inability to notify the end user would prevent the location being sent over the network.

⁹See the discussion on this subject in Variable Allocation [Weir and Noble, 2000].

also because the consequences of being out of disk are not as severe since the picture isn't lost. If this happens then the end user can find additional disk space in which to save the picture by inserting another memory card or deleting unnecessary files to make space.

- *File Server*

The File Server is started when a device boots up and is not stopped until the entire device shuts down. This is because the RAM used by the server is deemed an acceptable trade-off for providing quick responses to clients since it is a heavily used service.

Variants and Extensions

None known.

References

- *Fail Fast* (see page 17) describes an error-handling strategy often employed to deal with a failure to allocate the immortal resource during initialization.
- *Lazy Allocation* (see page 63) describes an alternative to this pattern.
- Variable Allocation [Weir and Noble, 2000] describes an alternative to this pattern.
- Eager Acquisition [Kircher and Jain, 2004] describes a variation of this pattern.

Lazy Allocation

Intent	Allocate resources, just in time, when they are first needed to improve how they are shared across the system.
AKA	Lazy Initialization, Lazy Allocation, On-Demand Loading, Deferred Loading

Problem

Context

The functionality of your application or service may require a resource but it is not needed immediately nor is it a critical requirement.

Summary

You have to resolve one or more of the following:

- You don't know if a resource will be needed when your application or service starts.
- The resources you are interested in are scarce and need to be shared across the system.
- There is a long period between your application or service starting and first use of the resource.
- You'd like to be able to start your component as quickly as possible.

Description

The temptation you'll face is to ensure that all the resources that may be needed for your component are allocated early in the lifetime of the component (e.g. at construction). This approach seems straightforward and is easy to understand because all the resources are allocated up front.

However, for resources which have a significant size, using *Immortal* (see page 53) can be problematic as the resources will not be available to others in the system. If a particular component allocates a large proportion of the available resources then all the time that the component is running it is preventing all other components in the system from using those resources.

Although the allocated resource usage constraints are a significant problem, the time taken to allocate a resource might be an additional challenge. While allocating one single resource, such as a single block of

memory or a single server connection, might take an insignificant time, many resource allocations can quickly add up to a significant amount of processing time or eat up a significant proportion of that type of resources in the whole system. This is particularly significant on a mobile device, where resources are constrained.

Even disregarding the extra power use, allocating resources when they might not be needed by the component can have an impact on the user experience. A user is not going to be happy with a maps application if it shows a loading screen for half a minute at start-up as it allocates the connection to the server to retrieve map information which is already cached on local media.

One way to solve this problem would be to use the Mayfly approach but this has the drawback of needing to be explicitly managed by the resource client. Ideally a solution would have the ease of use of *Immortal* (see page 53) but without preventing the resource from being shared until absolutely necessary.

Example

The Communication Infrastructure, known as Comms-Infras, within Symbian OS supports multiple protocol modules via a plug-in system. Each plug-in covers a different protocol, such as TCP/IP or the Bluetooth Audio Video Control Transport Protocol. The Communications Provider Module (CPM) interface is used to define each of these protocols. A straightforward approach to the lifetime of the resources needed by each protocol module would be to use *Immortal* (see page 53). However each protocol is generally non-trivial and can use 10s or 100s of kilobytes of RAM in addition to other resources. If an end user never makes use of one, or more, of these protocols, a likely scenario if their phone is the only device supporting Bluetooth that they own, then the resources used by such a protocol are effectively wasted. These resources would be more effectively used elsewhere.

For more details on Symbian OS communications, see [Campbell *et al.*, 2007].

Solution

Lazy allocation is based on the tactic of the resource client using a Proxy [Gamma *et al.*, 1994] to access the resource, rather than using the resource directly. The resource client allocates the Proxy, or lazy allocator as it is known here, immediately, as in *Immortal* (see page 53), for simplicity and delegates the actual management of the lifetime of the resource to the lazy allocator. This allows the lazy allocator to delay the allocation of the actual resource until the first time it is needed.

This pattern is written from the point of view of keeping a resource, once allocated, for as long as possible. This is because there are other

patterns that can be composed with this one that discuss de-allocation strategies; for simplicity, an *Immortal* (see page 53) approach is taken to de-allocation.

Structure

The resource, resource provider and resource client are as explained in the introduction to this chapter. The one additional object in this structure is the important part of this pattern. Instead of the resource client allocating the resource directly from the resource provider, a small Proxy object, the *lazy allocator*, is placed between the two (see Figure 3.3).

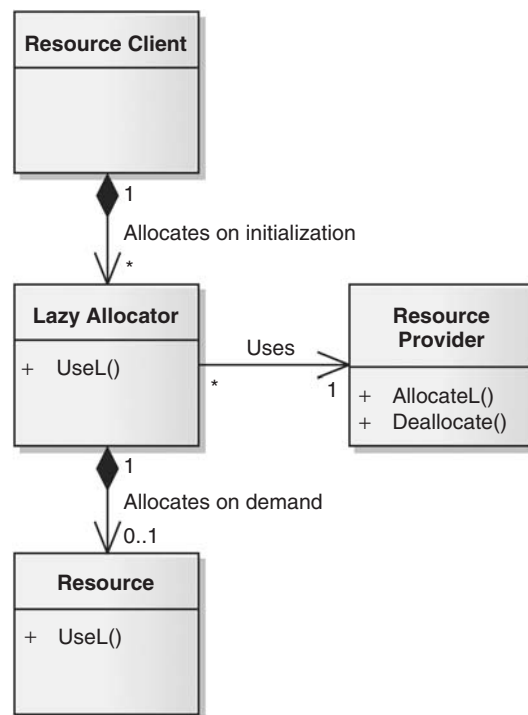


Figure 3.3 Structure of the *Lazy Allocation* pattern

The main responsibility of the lazy allocator is to delay the allocation of the resource from the resource provider until it is required. The lazy allocator provides the same interface as the resource itself so that the resource client interacts with the lazy allocator just as it would with the resource.

The resource client is expected to treat the lazy allocator as an immortal resource. The lazy allocator is then free to determine the best lifetime of the resource, removing the need for the resource client to check if the lazy allocator is ready for use.

Dynamics

When initialized, the resource client creates a lazy allocator object, which it uses for all the requests that would otherwise have been made directly on the resource (see Figure 3.4). It is only when the resource client first makes a call to the lazy allocator which requires the resource that the resource is actually allocated. Subsequent calls to the lazy allocator object are passed straight to the already allocated resource.

Implementation

In this chapter, C classes are used to demonstrate the implementation though it would be possible to use R classes instead with few changes. Also, for simplicity, error handling is done using *Escalate Errors* (see page 32) although an alternative error-handling strategy could be used.

Resource

The following code simply defines the resource we will be using:

```
class CResource : public CBase
{
public:
    // Performs some operation and Leaves if an error occurs
    TInt UseL();
}
```

Resource Provider

Here is the resource provider we will be using:

```
class CResourceProvider : public CBase
{
public:
    // Allocates and returns a new CResource object. If an error occurs,
    // it Leaves with a standard Symbian error code.
    static CResource* AllocateL(TInt aParam);

    // Matching de-allocator
    static void Deallocate(CResource* aResource);
}
```

Lazy Allocator

This Proxy object simply checks if its resource has been allocated, before calling the appropriate function on it. Since it is a Proxy, it must provide all the methods that the resource provides, in addition to the methods used to create the lazy allocator.

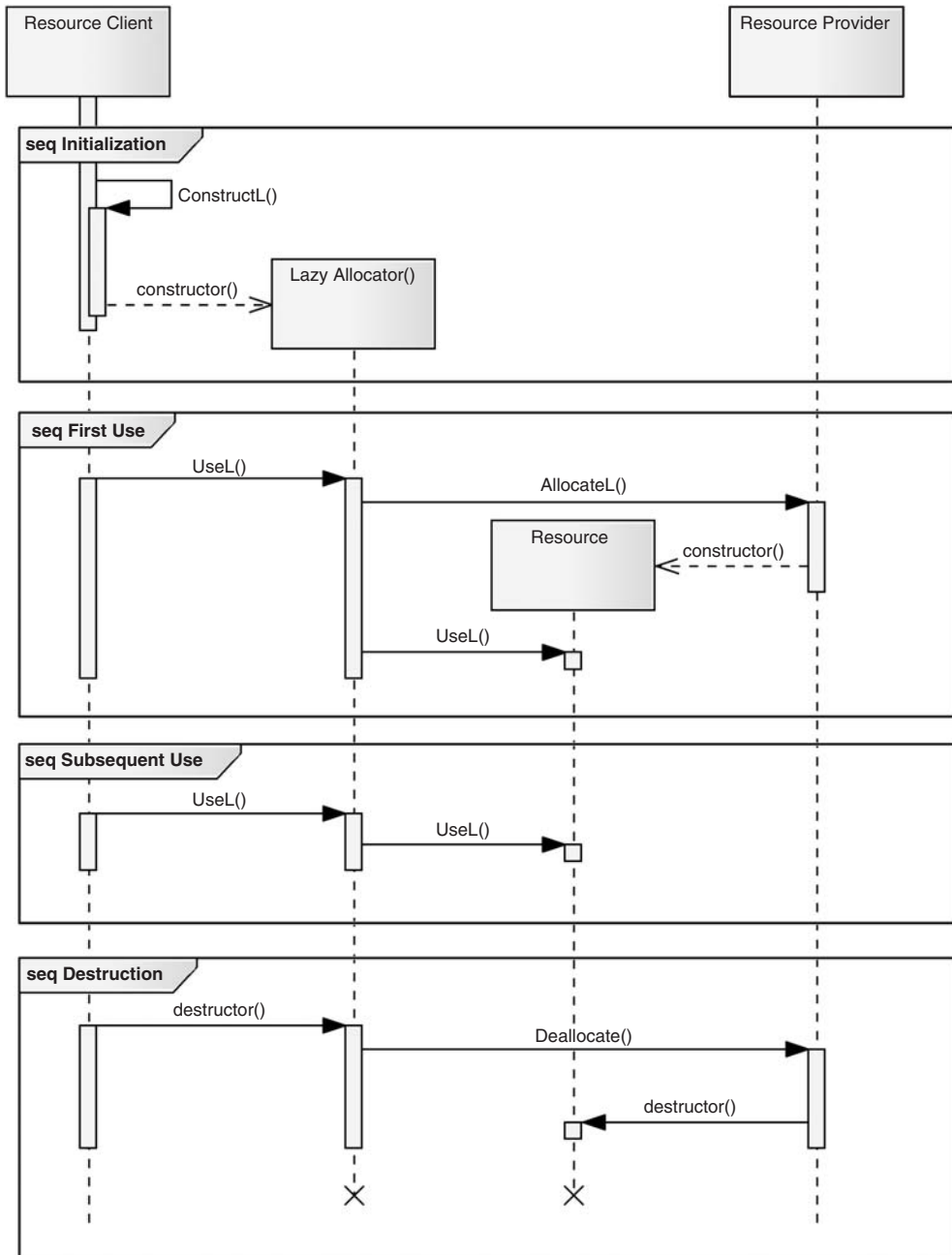


Figure 3.4 Dynamics of the *Lazy Allocation* pattern

```

class CLazyAllocator : public CBase
{
public:
    CLazyAllocator(TInt aAllocParam);
    ~CLazyAllocator();

    // CResource functions
    void UseL();

private:
    void InitializeResourceIfNeededL();

private:
    TInt iAllocParam;
    CResource* iResource;
};

```

The constructor needs to take the information needed to allocate the resource as parameters and remember it until later:

```

CLazyAllocator::CLazyAllocator(TInt aAllocParam)
    : iAllocParam(aAllocParam)
{
}

```

The destructor just needs to de-allocate the resource if it has been allocated. Note that `CResourceProvider::Deallocate()` should be used rather than deleting the resource directly since it allows the resource provider to choose how to deal with the resource. For instance, it may wish to keep the object and re-use it later.

```

CLazyAllocator::~~CLazyAllocator()
{
    if (iResource)
    {
        CResourceProvider::Deallocate(iResource);
    }
}

```

The private `InitializeResourceIfNeededL()` method is also straightforward. It first checks if the resource has been allocated and only allocates it if it hasn't:

```

void CLazyAllocator::InitializeResourceIfNeededL()
{
    if(iResource == NULL)
    {
        iResource = CResourceProvider::AllocateL(iAllocParam);
    }
}

```

The following implementation of `CLazyAllocator::UseL()` simply ensures the resource has been allocated before going on to use it:

```
void CLazyAllocator::UseL()
{
    InitializeResourceIfNeededL();
    iResource->UseL();
}
```

However, the problem comes when you are dealing with a resource function that doesn't return an error. The resource allocator functions have to be able to return an error since any one of them could be the first function called, attempt to allocate the resource and fail to do so. This means the resource allocator may not be able to provide a completely identical interface to the one provided by the resource.

Consequences

Positives

- Improves start-up time since resources are not allocated until they're needed. Since allocation occurs at the point of use, the time taken to allocate your resources occurs throughout the lifetime of your component, instead of occurring all at once.
- Reduces resource usage across the system since you only allocate the resources which are actually used.
- Simplifies the resource client by removing the need to check if the resource has been allocated before use.

Negatives

- Maintenance costs are increased because each use of the resource can fail and hence there are more error paths to deal with which makes your component more complex compared to using *Immortal* (see page 53).
- The cost of testing is increased due to the unpredictability of when the resource is actually allocated and the increased error costs compared to using *Immortal* (see page 53). Whilst it's true that the allocation is on first use, this doesn't provide much of a constraint on the behavior.
- This pattern is unlikely to be suitable for resources used in real-time operations as, for many resources, allocation is an unbounded operation. Use of a lazy allocator could trigger an allocation for any use of the resource and break your real-time constraints.

- The use of resources slowly grows over time since they are de-allocated only when the resource client decides the resource is no longer needed, which might be when the whole component is unloaded. However, this problem can be addressed using other patterns such as *Lazy De-allocation* (see page 73).

Example Resolved

Since Symbian OS v9.2, the Comms-Infras has supported the on-demand loading of CPMs under certain configurations.

During the initialization of Comms-Infras, an INI file is read for each CPM. The INI file specifies the CPM's configuration options including whether it should be loaded on demand. If the CPM is to be loaded on demand then a simple proxy object, `CModuleRef` (the lazy allocator), containing a pointer to the CPM is created. This lazy allocator object is added to the list of available CPMs within Comms-Infras. However, each CPM is only allocated and initialized when it is first used.

In this example, each CPM is its own resource provider as well as a resource.

Other Known Uses

This pattern is used throughout Symbian OS and here are just some of the many examples:

- *Image converters*
The Image Conversion Library (ICL) framework only loads specific converters for specific image types (GIF, JPEG, etc.) when the conversion of an image of the corresponding type is requested by a client.
- *Fonts*
The font and bitmap server only loads and renders fonts into memory when they are first requested. This is a particularly important use of this pattern when you consider that it is unlikely that all of the numerous fonts will be used all the time and that there might be fonts for different character sets (e.g. Latin, Greek or Kanji) that an end user might never use.
- *Transient servers*
These are only loaded when a client first connects to them. They are a common occurrence in Symbian OS. Examples are the Contacts Engine, Agenda Model and Messaging server, though whether they exhibit this behavior is configurable. For more information, see *Client-Server* on page 182.

Variants and Extensions

- *Idle Object*

This is a middle ground between immediate allocation of resources and lazy allocation. You start the process of resource allocation after or even during initialization of your component but only allow it to happen when your thread is not busy (see Figure 3.5).

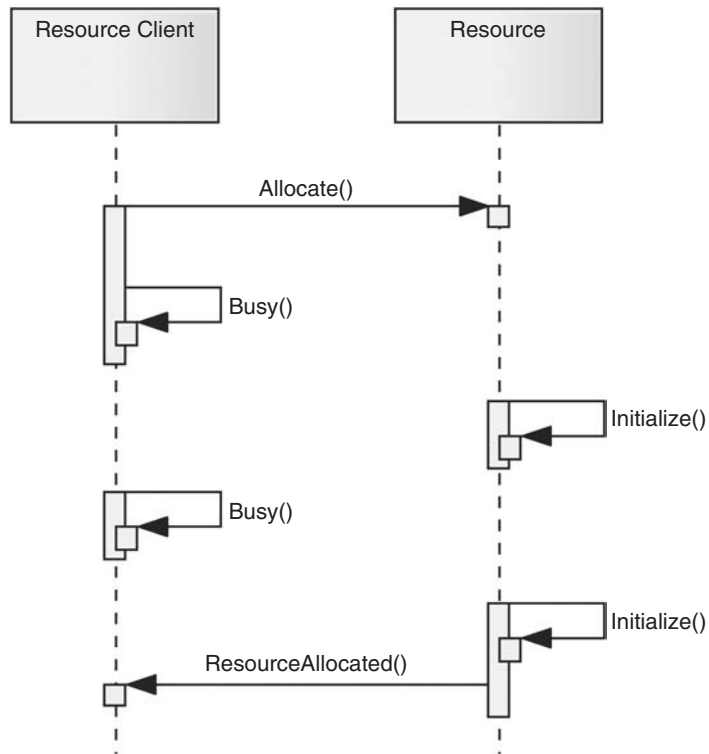


Figure 3.5 Dynamics of the *Idle Object* pattern

This is often used in situations where you know the non-essential resource is going to be used eventually but the resource client can start operating without it.

It is often implemented by using a `CIdle`-derived active object in situations where the creation and initialization of the resource is time intensive. The derivation from `CIdle` is a convenient way of de-prioritizing the initialization of the resource relative to your thread's other active objects.¹⁰

¹⁰`CIdle` works well here if you assign it the lowest possible active object priority.

References

- *Immortal* (see page 53) is an alternative resource lifetime pattern.
- *Lazy De-allocation* (see page 73) extends this pattern to de-allocate objects intelligently.
- *Escalate Errors* (see page 32) is used to handle allocation errors within the lazy allocator so that the resource client has the choice of whether to handle these errors or not.
- Proxy [Gamma *et al.*, 1994] is used to implement the lazy allocator.
- Lazy Acquisition [Kircher and Jain, 2004] covers similar ground within the context of other environments.

Lazy De-allocation

Intent Delay the de-allocation of a resource so that an almost immediate re-use of the resource does not incur a performance penalty.

AKA Delayed Unloading, Lazy Unloading

Problem

Context

A resource which is expensive to allocate or de-allocate is used frequently

Summary

- The resources you are interested in are scarce and need to be shared across the system.
- The use of the resource comes in bursts so that if it is used once there is a good chance it'll be used again soon.
- You need to provide quick responses and so need to avoid having to allocate a resource whenever possible.

Description

As has been repeated many times throughout this book, an open mobile device has a number of constraints that don't affect a PC; in comparison, a PC has almost limitless resources. Mobile devices are also different from small embedded systems such as microwaves and personal video recorders, which have similar, if not stricter, resource constraints. However, they are not expected to be multi-tasking nor to allow the installation of new applications or services which is a key requirement of an open device. Hence when you are developing software for Symbian you need to be a 'good citizen' and cooperate with the other components on a device by sharing resources well.

Part of being a 'good citizen' is to de-allocate resources that you have allocated when they are no longer needed. While this is most commonly true of allocations of memory, it is also true of other resources. For example, if your component allocates a radio communication channel and does not de-allocate the connection resource when it has finished using it then other components may be prevented from using the radio communication channel for a different purpose.

This understanding combined with a desire to make your application or service cooperate well with other software on the device should lead

you to consider a Mayfly approach and de-allocate resources as soon as they're no longer required. This frees them up to be re-used as soon as you have finished with them. The problem with this approach is that you might need the resource again almost immediately after you have finished using it.

Consider the situation where a user is using a web browser on their mobile phone to browse a website. The radio connection to the base station takes a few seconds to set up ('allocate') and then the download of the web page is relatively quick. Once the web page is downloaded, the web browser might immediately tear down ('de-allocate') the radio connection as it is no longer required. However as soon as the end user clicks on a link in the web page, the radio connection resource would need to be allocated again, leading to the end user waiting for a few seconds whilst the connection is re-established. This would have a negative impact on the end user's experience of the mobile web browser.

A possible solution to this problem is for the web browser developer to use *Immortal* (see page 53) and never de-allocate these resources. However, your component would stop being a 'good citizen' and prevent these resources from being shared across the system.

Example

Symbian OS contains a system for recognizing file types, such whether a file is a word-processed document or a specific type of audio or video file. This system is called the recognizers framework and it uses plug-ins to supply the code that identifies specific file types. Since there is a large number of different file types there is a corresponding large number of recognizer plug-ins.

When loaded, recognizers use a significant amount of RAM in addition to taking time to initialize during device start-up so using *Immortal* (see page 53) wouldn't be appropriate.

However, the Mayfly strategy would also not be appropriate because recognizers tend to be used in quick succession over a short period such as when a file browser enumerates all the file types in a directory. Hence, using Mayfly would result in file recognition taking longer because the recognizers would be allocated and de-allocated many times.

Solution

This pattern is based on the tactic of the resource client using a Proxy [Gamma *et al.*, 1994] to access the resource and the resource provider, called the *lazy de-allocator*. The resource client takes ownership or

releases the underlying resource as frequently as is needed¹¹ but it delegates the management of the actual lifetime of the resource to the lazy de-allocator, which is never de-allocated. This allows the lazy de-allocator to delay the actual de-allocation of the resource.

The idea is that when the resource is no longer required by the resource client, indicated by it releasing the resource, then instead of it being immediately unloaded it is only unloaded when a specific condition is met. This is done in the expectation that there is a good chance the resource is going to be needed again soon so we shouldn't unnecessarily de-allocate it since we'd just have to allocate it again.

This pattern is written from the point of view of immediately allocating a resource in the expectation that it'll be needed almost straight away. There are other patterns that can be composed with this one and that discuss allocation strategies so, for simplicity, an *Immortal* (see page 53) approach is taken to allocation.

Structure

The responsibilities of the objects shown in Figure 3.6 are as follows, in addition to their description in the chapter introduction:

- The resource client creates the lazy de-allocator as soon as it is created and then uses the `PrepareResourceL()` and `ReleaseResource()` functions to indicate when it is using the lazy de-allocator as a resource.
- The lazy de-allocator provides both the resource provider and resource interfaces to the resource client as well as managing the actual lifetime of the resource by choosing when to de-allocate it.

Dynamics

Figure 3.7 shows the resource being continually used and never actually being de-allocated by the lazy de-allocator.

In Figure 3.8, the resource is used only infrequently and so the lazy de-allocator has decided to de-allocate it so that it can be shared across the system. The way that the lazy de-allocator works is as follows:

- Each time `ReleaseResource()` is called on the lazy de-allocator it calls `AddDeallocCallback()`. This is the point where the lazy de-allocator decides whether or not to create a callback that will fire at some point in the future to de-allocate the resource.

¹¹Using the Mayfly approach.

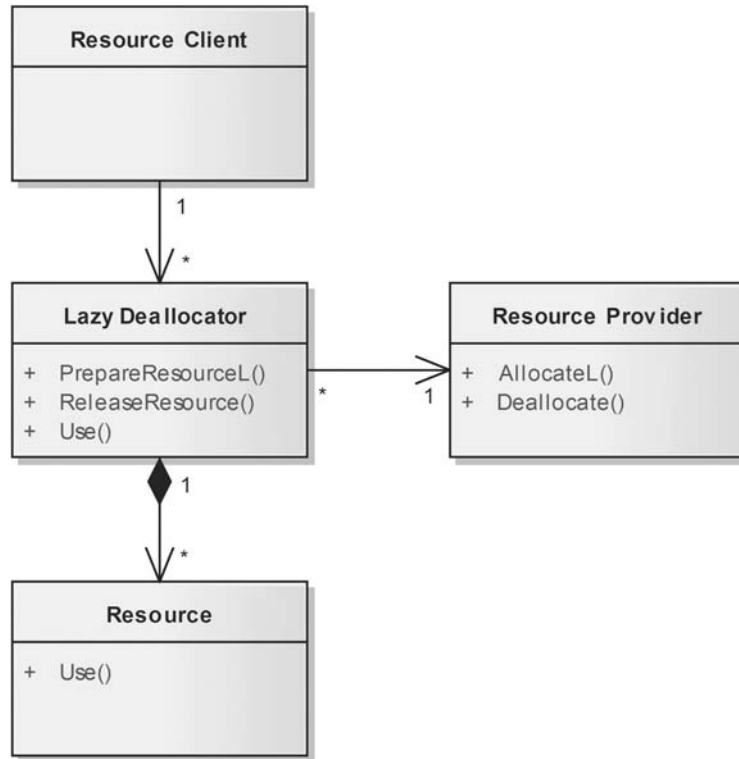


Figure 3.6 Structure of the *Lazy De-allocation* pattern

- Each time `PrepareResourceL()` is called on the lazy de-allocator it calls `RemoveDeallocCallback()` to prevent the callback from firing so that the resource remains allocated.

The most important thing that you need to decide when applying this pattern is how to implement the `AddDeallocCallback()` function as this will be a key factor in determining the trade-off between avoiding the execution cost of allocating an object and sharing the resource with the rest of the system. Some common strategies include:

- A timer could be used to call `DeallocCallback()`. This timer would be started in `AddDeallocCallback()` and stopped in `RemoveDeallocCallback()`.
- If the lazy de-allocator is used as a Proxy for multiple resources of the same type, then `AddDeallocCallback()` might simply ensure that it maintains a constant pool of unowned but allocated resources. This strategy is very similar to Pooled Allocation [Weir and Noble, 2000].

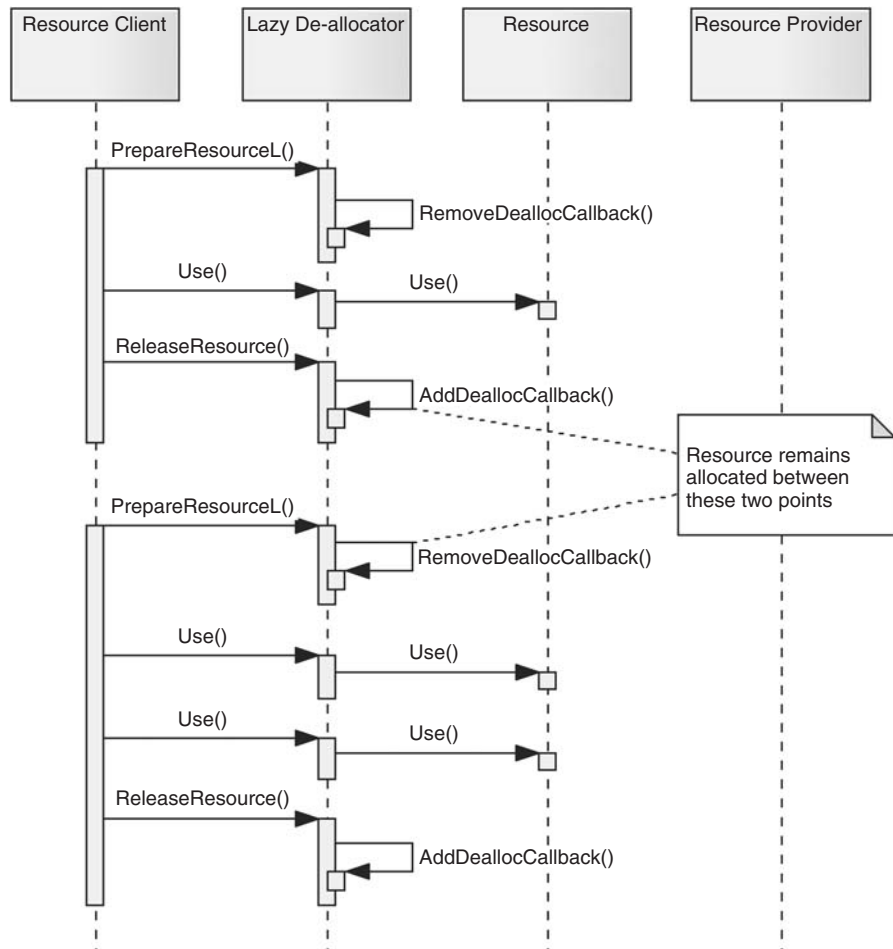


Figure 3.7 Dynamics of the *Lazy De-allocation* pattern when the resource is in frequent use

Implementation

Some notes before we get started:

- Just to illustrate the code given here a timer has been chosen as the de-allocation strategy.
- For simplicity, the lazy de-allocator only handles a single resource but it could easily be extended to handle more than one.
- For simplicity, error handling is done using *Escalate Errors* (see page 32) although an alternative error-handling strategy could be used.
- C classes are used to demonstrate the implementation though it would be possible to use R classes instead with few changes.

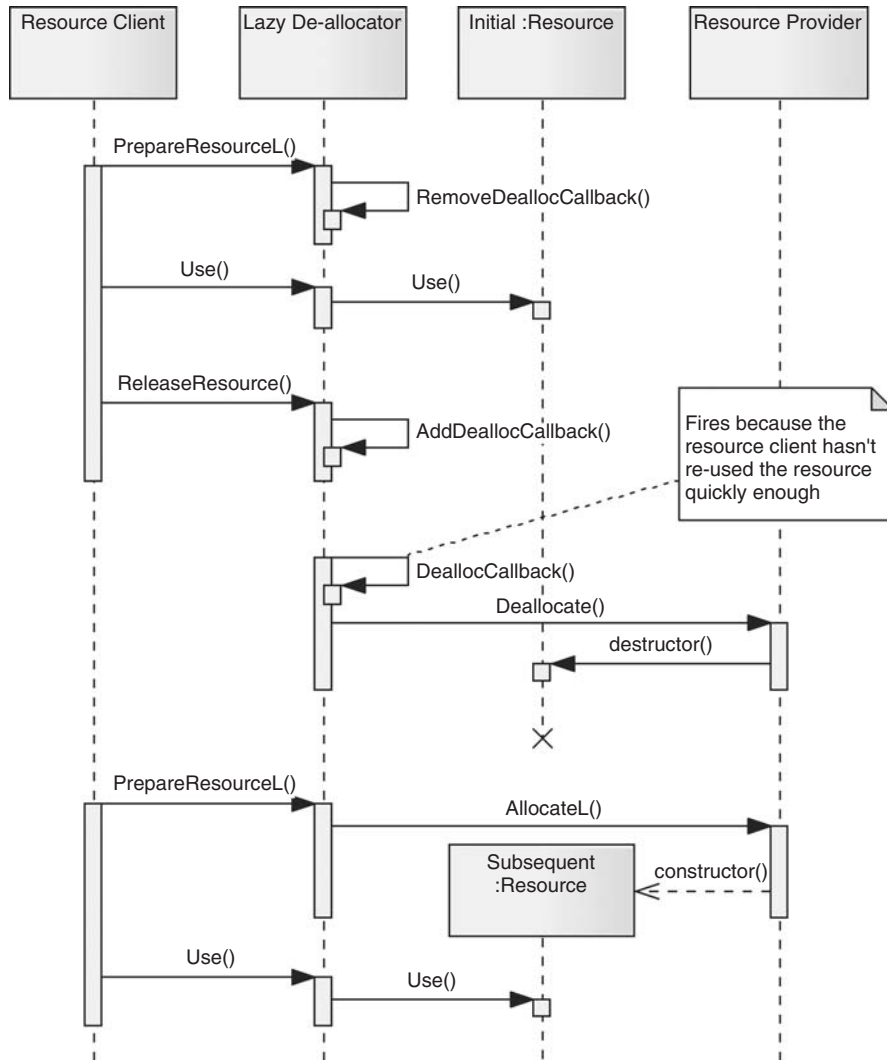


Figure 3.8 Dynamics of the *Lazy De-allocation* pattern when the resource is not in frequent use

Resource

The following code simply defines the resource we are using:

```

class CResource : public CBase
{
public:
    // Performs some operation and Leaves if an error occurs
    void UseL();
}
  
```

Resource Provider

Here is the resource provider:

```
class CResourceProvider : public CBase
{
public:
    // Allocates and returns a new CResource object. If an error occurs,
    // it Leaves with a standard Symbian error code.
    static CResource* AllocateL();

    // Matching De-allocator
    static void Deallocate(CResource* aResource);
};
```

Lazy De-allocator

This class makes use of the standard Symbian OS CTimer class¹² to handle the de-allocation callback:

```
class CLazyDeallocator: public CTimer
{
public:
    CLazyDeallocator* NewL();
    ~CLazyDeallocator();

public: // Resource provider i/f
    // Ensure a resource is available for use or Leave if that isn't
    // possible
    void PrepareResourceL();

    // Signal that the client isn't using the resource any more
    void ReleaseResource();

public: // Resource i/f
    void UseL();

private:
    CLazyDeallocator();
    void ConstructL();

    // From CActive
    void RunL();
    TInt RunError(TInt aError);

private:
    CResource* iResource;
};
```

The constructor and destructor of this class are straightforward implementations. One point to note is that, to avoid disrupting other more

¹²This is implemented in terms of *Active Objects* (see page 133).

important tasks, a low priority is given to the timer and hence to when the de-allocation callback is run:

```
CLazyDeallocator::CLazyDeallocator()
: CTimer(CActive::EPriorityLow)
{
}

// Standard two-phase construction - NewL not shown for conciseness.
void CLazyDeallocator::ConstructL()
{
    iResource = CResourceProvider::AllocateL();
}

CLazyDeallocator::~~CLazyDeallocator()
{
    if (iResource)
    {
        CResourceProvider::Deallocate(iResource);
    }
}
```

The method for retrieving a resource needs to cancel the de-allocation timer and ensure that a resource is available for later use:

```
CResource* CLazyDeallocator::PrepareResourceL()
{
    Cancel(); // Stops the de-allocation callback

    // Ensure we have a resource for use later
    if (!iResource)
    {
        iResource = CResourceProvider::AllocateL();
    }
}
```

The following function, `ReleaseResource()`, simply starts the timer for the `iResource` de-allocation callback:

```
const static TInt KDeallocTimeout = 5000000; // 5 seconds

void CLazyDeallocator::ReleaseResource()
{
    // Check if we're already active to avoid a stray signal if a client
    // incorrectly calls this function twice without calling
    // PrepareResourceL() in between
    if (!IsActive())
    {
        After(KDeallocTimeout);
    }
}
```

The value to use for the de-allocation timeout is dependent on the situation in which you are using this pattern. To decide upon the best value for the timeout you should first consider the use cases for the

allocation and de-allocation of the resources by obtaining logs showing resource allocation and de-allocation to observe your usage patterns. After this, an informed decision can be made about the value to use for the timeout. If it is too long, such as hours perhaps, then the resource might never be de-allocated, but if it is too short, such as a millisecond, then the resource may be de-allocated after every use.

By isolating the timeout value into a single constant, it is very straightforward to change the timeout value at a later stage. You are likely to want to be able to change the timeout once the dynamic behavior has been observed in a running system or to fine tune future versions of your software. Isolating the timeout into a single constant makes this adjustment far easier. You might even find it useful to determine the timeout when constructing the lazy de-allocator at run time, such as by reading a value from the Central Repository.

The following implementation of `CLazyDeallocator::Use()` simply forwards the function call on to the resource:

```
void CLazyDeallocator::Use()
{
    // Panic if the client hasn't called PrepareResourceL()
    ASSERT(iResource);

    iResource->Use();
}
```

When the timer expires, the `CLazyDeallocator::RunL()` implementation is run to de-allocate the resource:

```
void CLazyDeallocator::RunL()
{
    User::LeaveIfError(iStatus.Int());

    CResourceProvider::Deallocate(iResource);
    iResource = NULL;
}
```

The implementation of `CLazyDeallocator::RunError()` is left out for clarity. No `CLazyDeallocator::DoCancel()` function is needed either, since it's provided by `CTimer`.

Resource Client

Whenever the resource client is first constructed, it creates itself a `CLazyDeallocator` object for use later:

```
void CResourceClient::ConstructL()
{
    iResourceProxy = CLazyDeallocator::NewL();
}
```

However, when the resource client actually wants to use the resource, it does so by letting the `CLazyDeallocator` object know when and for how long by calling first `PrepareResourceL()` and then `ReleaseResource()`. The resource client should use these functions as frequently as possible so that the `CLazyDeallocator` object is able to keep an accurate track of when the underlying resource is actually being used:

```
void CResourceClient::DoOperationL()
{
    iResourceProxy->PrepareResourceL();

    // Perform required actions with resource
    iResourceProxy->Use();
    ...

    iLazyUnloader.ReleaseResource();
}
```

Consequences

Positives

- There is better sharing of resource within the system when compared with more ‘greedy’ allocation strategies such as *Immortal* (see page 53).
- Less time is wasted in repeatedly allocating frequently used resources – one resource is allocated and reused, allowing your component to be more responsive.
- Your software is simplified – the responsibility for optimizing the resource lifetime is in one place rather than spread out across a whole component.
- The resource client is still able to control when a resource is available, and hence the (de-)allocation error paths, by calling `CLazyDeallocator::PrepareResourceL()`.

Negatives

- Resource use is unpredictable – there is no direct control over the number of resources allocated at one time which could lead to a resource shortage while waiting for the de-allocation callbacks for unused resources to run.
- Testing a component using this pattern may become more difficult because of the unpredictability of its resource usage.
- The de-allocation callback uses additional resources – in the implementation above, this was the timer. This pattern is only good for situations where the resources being managed are significantly more expensive than the resources needed for the de-allocation callbacks.

- This pattern is not suitable if the resource is very frequently used – in these cases the de-allocation callback will rarely, if ever, expire as it will be reset after every use. This leads to extra complexity for no gain as the resource is never out of use for long enough to be de-allocated. For this situation, *Immortal* (see page 53) would be more suitable.

Example Resolved

The recognizer framework uses this pattern to solve the problem of how to handle the allocation and de-allocation of individual recognizer plug-ins. This was introduced in Symbian OS v9.1 as a way to reduce the RAM used by a device as well as providing performance optimization so that the recognizer framework, and hence the device, would start up more quickly. To preserve compatibility, this behavior was made configurable at ROM build time.

The recognizer framework uses a timer as its strategy for determining when to run the de-allocation callback with a timeout of 10 seconds by default. However, because the use of recognizers will vary from device to device, this timeout was also made configurable at ROM build time.

By applying this pattern, it was possible to solve both problems: freeing the resources used by the recognizers for the majority of the time while preserving an acceptable level of performance in use cases where the recognizers are heavily used.

Other Known Uses

- *Transient servers*
Many servers are only started when needed by implementing this pattern. Instead of closing completely when the last client disconnects a timer is started. If a new connection to the server happens during the timer period then it is canceled and the server's destruction is aborted. The messaging server is one such example.¹³
- *Audio Video Distribution Transport Protocol*
The AVDTP uses this pattern to avoid the high cost of tearing down and then re-establishing a Bluetooth connection unless absolutely necessary. This is done by using a timer that is only started once the last client of the protocol has disconnected.

Variants and Extensions

- *Automatic Lazy Resource Proxy*
In this variant, the resource client doesn't tell the lazy de-allocator when it is using a resource by calling `PrepareResourceL()` or

¹³In some configurations.

`ReleaseResource()`. Instead, it simply calls one of the resource functions that the lazy de-allocator provides as a proxy for the resource and the lazy de-allocator ensures a resource is available to meet the request.

This is done by checking if the resource is allocated at the start of each resource operation exposed by the lazy resource and starting a de-allocation callback at the end of every method, which would look something like this:

```
void CLazyResource::UseL()
{
    if(!iResource)
    {
        iResource = CResourceProvider::AllocateL();
    }

    // Perform requested action
    iResource->Use();

    // Start the timer for the iResource de-allocation callback
    if (!IsActive())
    {
        After(KDeallocTimeout);
    }
}
```

This approach combines both *Lazy Allocation* (see page 63) and *Lazy De-allocation* into a single pattern.

- *Pooled Allocation* [Weir and Noble, 2000]
In this variant, a group of resources is allocated by a resource provider as soon as it is created. These resources are owned by the resource provider until a resource client makes an allocation. This allocation is immediately satisfied from the resource pool, if a resource is available, otherwise a full allocation occurs to satisfy the request. When a resource is de-allocated, it is returned to the resource pool unless the pool has already reached a fixed size, at which point it is destroyed. It is only when the resource provider itself is destroyed that all the resources in the resource pool are de-allocated.

References

- *Immortal* (see page 53) describes an alternative approach to managing resource lifetimes.
- *Lazy Allocation* (see page 63) is often combined with this pattern.
- *Pooled Allocation* [Weir and Noble, 2000] describes a variation of this pattern.

- Pooling [Kircher and Jain, 2004] is a pattern similar to Pooled Allocation.
- Leasing [Kircher and Jain, 2004] is a pattern that allows a Resource Lifecycle Manager to specify the time for which the resources are available.
- Evictor [Kircher and Jain, 2004] is a pattern that allows for the controlled removal of less frequently used resources from a cache.

4

Event-Driven Programming

Power conservation is important in mobile devices since battery life is usually one of the top three reasons a consumer will choose a particular device. Mobile software must be designed to be frugal in its use of hardware components that consume power, such as the CPU, display and radio units. These *power-sink* hardware components usually have various modes of operation including low-power modes which can be used to minimize power drain whenever the hardware components, or software services related to them, are not in use. An example of a power-sink component and a related service might be a Bluetooth radio unit and a Bluetooth device discovery service. Such hardware components are often encapsulated as shared resources in Symbian OS and are managed on behalf of the programs that use them (often using a reference counting mechanism).

One power-sink hardware component that all programs utilize is the CPU. The software running on a mobile device has a direct influence upon the power consumed by the CPU. This is because the CPU uses power for each CPU cycle executed. The fewer CPU cycles needed to execute a piece of functionality, the less power the device uses. This applies to all types of software, from device drivers for hardware resources to user-level applications. All Symbian developers should understand and apply these patterns throughout their software design and development.

Consider an indicator on a mobile phone that shows whether a Bluetooth communication link is active by displaying one icon when it is in use and another icon when it is not. One approach to keeping this icon updated would be to have a thread running that periodically requests information from the Bluetooth subsystem and updates the icon accordingly. This periodic approach is generally known as *polling*.

The difficulty with polling is that, while it is easy to write, every time the monitoring thread wakes up and requests information from the Bluetooth subsystem, it causes the CPU to be woken up and made to do work, whether the icon needs to be updated or not. This is clearly wasteful of

CPU time. Worse, because this thread would be running all the time,¹ the cumulative effect of this thread running could be a marked reduction in battery life.

Arguably, this could be ameliorated by lengthening the period between updates, but this would then worsen the other drawback of polling – that it can produce unresponsive software. For instance, if the Bluetooth subsystem became inactive just after the icon had been redrawn, it would take at least an entire polling period before the display is updated.

The alternative to polling is ‘event-driven programming’, which allows the CPU and other hardware components to be used in the most power-efficient and responsive way.

Fundamentally, there are four concepts in event-driven programming:

- An *event* is some significant occurrence. In the Bluetooth icon example, a link becoming active or being disconnected is an event. Examples of events in the user interface domain are keystrokes, screen clicks and screen redraws.
- An *event signal* is an indication that an event has occurred. Events and the resulting event signals also occur in middleware and low-level software such as when hardware data buffers have been filled with incoming data, an email has been sent or a USB cable has been attached. Event signals also drive activity at the hardware level where they are more commonly known as *interrupts*. For the Bluetooth icon example, the event signal is the notification sent by the Bluetooth subsystem about the new state of the Bluetooth link.

The amount of information carried in an event signal can vary from nothing more than an indication that a particular event has happened (a *warm* event signal) to a comprehensive description of the event (a *hot* event signal). In the former case, the event consumer may then need to make an additional effort to discover details about the event that occurred. In the latter case, the event signal can be considered self-contained.

- An *event consumer* is an entity which requests notification of a particular event or type of event from an event generator. For the Bluetooth icon example, the event consumer is the UI component that owns the Bluetooth icon.
- An *event generator* is an entity which produces events and sends event signals to registered event consumers. For the Bluetooth icon example, the event generator is the Bluetooth subsystem. Some event generators produce events as a result of stimuli in the environment, such as a keyboard driver producing keystroke events, while others generate events prompted by the action of the event consumer

¹Potentially even when the user is not operating the phone.

requesting a notification. For instance, a communications stack sends an event indicating the establishment of a connection to the client that requested the connection.

It is worth noting that many components operating in an event-driven environment are both event consumers and event generators. Such event consumer–generators may be linked in long chains or networks to achieve some overall task, known as an *event-driven chain*.

Figure 4.1 illustrates the following event-driven chain:

- A piece of communications hardware (an event generator) generates an interrupt when data is received (an event) that is picked up by a device driver (an event consumer).
- A communications stack (an event consumer) is informed by a device driver (an event generator) that some data has been received (an event).
- The communications stack (an event generator) then determines if the new data is bound for one of its clients (an event consumer) and, if so, sends it a notification (an event signal).
- The messaging application client (an event generator) determines that the data is a new message and informs its UI (an event consumer) that a new message has arrived (the event).

Note how the CPU goes into low power mode between processing the various event signals. Of course in a full system there could be other activities that keep the CPU from going into a low power mode for at least some of the time between the steps in this sequence.

In addition to being more power efficient and responsive, event-driven programming helps you to decouple your components. This is because you are able to distribute responsibility for generating or consuming events as needed to the appropriate components because an event can be almost anything and exist almost anywhere. In addition an event generator doesn't need to know about the consequences of the event signal nor does an event consumer need to worry about where the event signal has come from. In the Bluetooth icon example, the Bluetooth subsystem doesn't need to worry about any dependencies on the higher UI layer. It just provides the event signal and carries on regardless of what is done with it by the Bluetooth icon component.

Symbian OS has a number of classes and common idioms that support event-driven programming which can be adopted by developers into the design of their own components.

One of the most common event-driven programming tools in Symbian OS is the use of active objects, commonly known as AOs. They are described by the *Active Objects* pattern (see page 133).

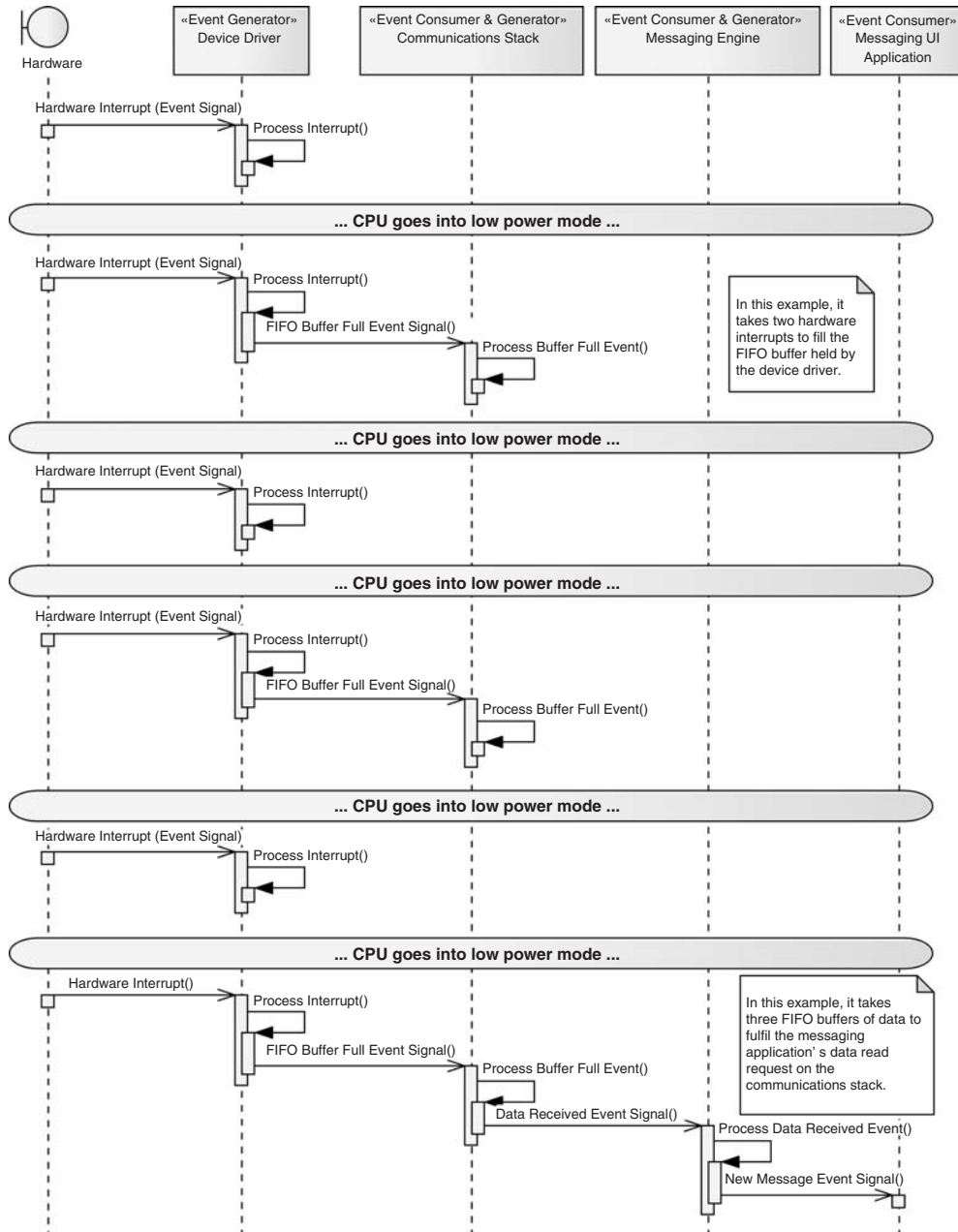


Figure 4.1 An event-driven chain

It is worth considering that, in event-driven programming terms, an AO is an event consumer; the `CActive` class contains a `TRequestStatus` member variable which, when an AO is activated, is passed to some asynchronous function, putting the AO into the role of event consumer. This asynchronous function then acts as the event generator. The event signal is the completion of this `TRequestStatus`, which triggers the AO's `RunL()` function.

Aside from AOs, there are a number of other common event-driven programming idioms used in Symbian OS. These are represented as the following patterns:

- *Event Mixin* (see page 93)
- *Request Completion* (see page 104)
- *Publish and Subscribe* (see page 114)

The following list summarizes the main characteristics of these patterns to help you to understand when to use each of them.

Event Mixin:

- Both the event generator and the event consumer must be in the same thread.
- There is one event generator and one event consumer per event signal.²
- Both warm and hot event signals are supported.
- Reliable but not secure event signals are supported.³
- The event generator knows whether an event consumer is listening.

Request Completion:

- The event generator and the event consumer need not be in the same thread.
- There is one event generator and one event consumer per event signal.
- Both warm and hot event signals are supported.
- Reliable but not secure event signals are supported.
- The event generator knows whether an event consumer is listening.

²*Event Mixin* (see page 93) is, by default, one-to-one but can be extended to support multiple event generators and event consumers.

³When each event consumer can be guaranteed to receive each and every event signal, it is known as *Reliable event signals*.

Publish and Subscribe:

- The event generator and the event consumer need not be in the same thread.
- There can be zero to many event generators and zero to many event consumers per event signal.
- Secure but not reliable event signals are supported.
- The event generator does not know whether an event consumer is listening.

Event Mixin

Intent Save power and decouple your components by defining an interface through which event signals are sent to another component in the same thread.

AKA Observer [Gamma *et al.*, 1994]

Problem

Context

You wish to propagate event signals synchronously between an event generator and an event consumer operating within the same thread.

Summary

- You wish to reduce the power usage of your component.
- You want to promote encapsulation of and loose coupling between your event generators and event consumers so that testing and maintaining your component is made easier.

Description

An event generator wishes to propagate event signals to an event consumer that is running in the same thread. One way to do this would be to enable the event generator to make calls directly to the event consumer object's member functions. However, this promotes poor encapsulation (the event generator is able to call functions of the event consumer that are unrelated to event signaling) and tight coupling (the event generator is hardcoded to generate events for this specific event consumer).

Example

Object EXchange (OBEX) is a communications protocol that defines a means for two devices to communicate by passing *objects* between them. The protocol defines the two ends of an OBEX communication session as the OBEX client and the OBEX server. The OBEX client drives communication by sending requests⁴ to the OBEX server. The OBEX server is expected to respond to these requests when it receives them.

A single device can support multiple OBEX servers, each offering a different service to the remote OBEX client such as file transfer or

⁴Such as 'get an object'.

contact details transfer. Each OBEX service requires its own specific logic in addition to the generic OBEX server logic. Hence, to support OBEX servers, a way is needed to separate the generic handling of the OBEX protocol, forming and parsing packets, from the service-specific logic offered by the OBEX server in question. The generic logic, as an event generator, would need to be able to send event signals to the service-specific logic, as an event consumer, when a request from the client is received. It is also necessary to allow the generic logic to be bound to a number of different instances of service-specific logic in order to satisfy the requirement for multiple OBEX services to run on the same device at the same time.

In order to achieve these design goals, one possible solution would have been to use function pointers but this brings with it a host of problems such as non-intuitive syntax and code that is difficult to read and debug.

Solution

A *mixin* is an abstract interface that provides a certain functionality to be inherited by a class. Classes like this in Symbian OS have their names prefixed by an 'M' and are known colloquially as *M classes*. Inheritance of an M class effectively extends the API of the derived class and, through the use of pure virtuals, can force a derived class to provide an implementation before it can be instantiated.

The main thrust of this pattern is that the developer of the event generator should define an M class, or *event mixin*, for event consumers to derive from. The event generator then sends event signals by calling functions on the event mixin.

Structure

Figure 4.2 shows how this pattern is structured. In C++, an M class is implemented as a class that contains at least one pure virtual function⁵ and no data members. Their lack of member data allows them to be easily inherited by classes that already inherit from other classes, known as *multiple inheritance*, but avoids the problem of a class ending up containing multiple copies of a member variable, particularly where a class inherits from two classes sharing a common subclass.

However, with multiple inheritance it is also possible to have more than one function with the same signature. This problem is reduced by the following naming convention: prefix the initials of the class name to the name of the function. In Figure 4.2, this is shown by the `MEventMixin` function being called `MemEventHasOccurredL()` rather than `EventHasOccurredL()`.

⁵It often contains just pure virtual functions.

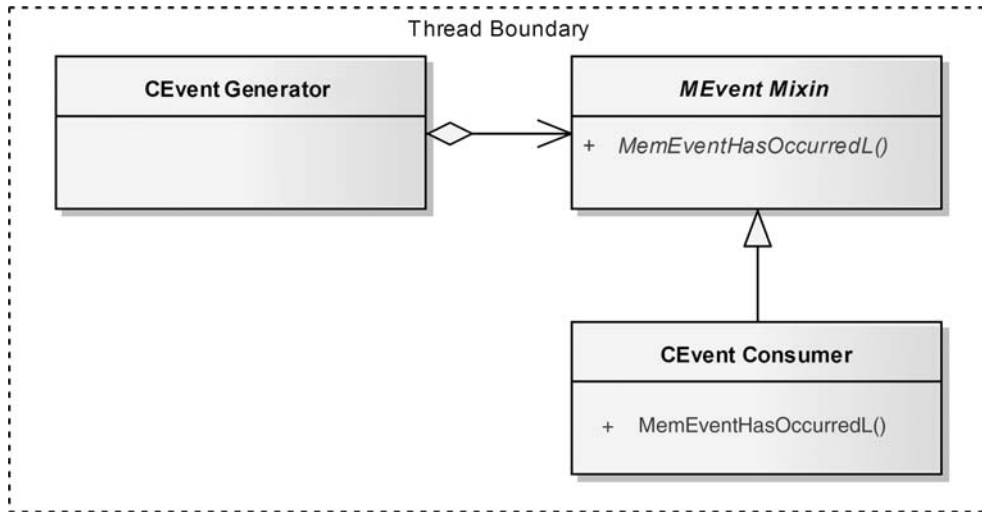


Figure 4.2 Structure of the *Event Mixin* pattern

Dynamics

Before any event signals can be propagated, the event generator must be provided with a pointer or reference to the event consumer. How this occurs depends very much on the relationship between the event generator and event consumer, the relative life spans of the two objects, and the number of event consumers that the event generator can support.

A couple of common approaches are listed below:

- The event consumer already exists so the event consumer itself creates the event generator, passing a reference to itself into the event generator's factory function.
- The event generator already exists so the event consumer calls a registration function provided by the event generator, passing in a reference to itself.

Once the event consumer has been registered with the event generator, an event signal can be produced by the event generator calling the appropriate member function of the mixin. The event consumer then takes action, before returning program control to the event generator. A potential problem arises here⁶ because the event generator is calling the event consumer synchronously and hence is blocked until the function call returns. This means a badly behaved event consumer can break an event generator. If this is a problem for you, then you can further decouple

⁶Especially if the event consumer is provided by a different vendor from the event generator.

the two by making the call asynchronous, such as by using *Active Objects* (see page 133).

Finally, when no longer concerned with receiving event signals, the event consumer may de-register its interest. The means by which it does this will usually mirror the means by which the original registration took place, for example:

- The event consumer is longer lived and hence deletes the event generator.
- The event consumer is longer lived and hence calls a deregistration function provided by the event generator, passing in a pointer to itself as a way of identification.

It is a common requirement for event mixin functions to be able to handle errors since they actually do things that might fail. This pattern is illustrated using *Escalate Errors* (see page 32), as indicated by the trailing 'L' on the event mixin function names, although alternative strategies are possible.

Implementation

The following code shows a very simple class as an event generator, using an event mixin as the API through which it notifies an event.

Event Generator

To illustrate this pattern the `MEventMixin` class given here provides only one possible event type, `MemEventHasOccurredL()`, and provides very little contextual information on the nature of the event.⁷ However, you will often find that an event mixin class is used as a means to convey a number of events and so contains multiple functions which may pass contextual information as parameters to further refine the nature of event.

```
class MEventMixin
{
public:
    virtual void MemEventHasOccurredL() = 0;
};
```

The following is a definition of an event generator. Note that it is normally defined at the same time as the `MEventMixin` class although often in a separate header file so that the event consumer can just include the `M` class.

⁷Hence it is a warm, rather than a hot, event signal.


```

class MEventMixin; // Forward declare

class CEventGenerator : public CBase
{
public:
    void RegisterEventConsumerL(MEventMixin& aEventConsumer);
    void DeregisterEventConsumerL(MEventMixin& aEventConsumer);

private:
    void ActivityComplete();

private:
    MEventMixin* iRegisteredEventConsumer;
};

```

The following implementation allows a single event consumer to be registered although this could be extended to allow multiple event consumers fairly easily. To protect ourselves from misuse of the API by clients we should check to see if an event consumer is already registered:

```

#include "MEventMixin.h"

void CEventGenerator::RegisterEventConsumerL(MEventMixin& aEventConsumer)
{
    if (iRegisteredEventConsumer)
    {
        User::Leave(KErrInUse);
    }
    iRegisteredEventConsumer = &aEventConsumer;
}

```

The following code allows an event consumer to be de-registered. To protect ourselves from misuse of the API by clients, we should check to see if the event consumer passed in is actually the one registered:

```

void CEventGenerator::DeregisterEventConsumerL(MEventMixin&
                                                aEventConsumer)
{
    // Check the correct event consumer has been passed in
    if (iRegisteredEventConsumer == &aEventConsumer)
    {
        iRegisteredEventConsumer = NULL;
    }
    else
    {
        User::Leave(KErrArgument);
    }
}

```

The private function is called when the event consumer has an event signal it wishes to send and shows how the `MEventMixin` is used:

```
void CEventGenerator::ActivityComplete()
{
    if(iRegisteredEventConsumer)
    {
        iRegisteredEventConsumer->MemEventHasOccurredL();
    }
}
```

Event generators should take care to deal with the event consumer calling back into the event generator within their implementation of `MemEventHasOccurredL()`.

Event Consumer

The classes above could then be used by another class acting as an event consumer as follows. We show here an implementation based on an event generator that exists both before and after the lifetime of the event generator:

```
#include "MEventMixin.h"
class CEventGenerator;

class CEventConsumer : public CBase, public MEventMixin
{
public:
    CEventConsumer* NewL(CEventGenerator& aEventGenerator);
    ~CEventConsumer();

public: // From MEventMixin
    void MemEventHasOccurredL();

private:
    CEventConsumer(CEventGenerator& aEventGenerator);
    void ConstructL();

private:
    CEventGenerator& iEventGenerator;
};
```

Note that the event consumer class has a pointer to the event generator class purely to allow it to register itself as an event consumer. This could be removed if the event consumer was registered by some notional management class owning pointers to both the `CEventGenerator` and `CEventConsumer` objects.

Here is the implementation of the event consumer class:

```
#include "CEventGenerator.h"
#include "CEventConsumer.h"

CEventConsumer::CEventConsumer(CEventGenerator& aEventGenerator)
    : iEventGenerator(aEventGenerator)
{
}
```

The following code shows the event consumer registering with the event generator immediately although this could be done later if required. Note that the standard `NewL()` implementation is not shown here for brevity.

```
void CEventConsumer::ConstructL()
{
    iEventGenerator.RegisterEventConsumerL(*this);
}
```

Event consumers must be careful to de-register from the event generator when notifications are no longer needed. Here we show it being done during the destruction of the event consumer but it could be done earlier.

```
CEventConsumer::~CEventConsumer()
{
    iEventGenerator.DeregisterEventConsumerL(*this);
}
```

The implementation of the event mixin functions depend on how this pattern is being used in your specific solution:

```
void CEventConsumer::MemEventHasOccurredL()
{
    // Do some processing in response to the event
}
```

Consequences

Positives

- This pattern reduces the power usage of your components.
- This pattern is simple to implement and provides considerable flexibility:
 - Event generators know when there are event consumers listening for event signals.

- Event generators can send event signals by calling a mixin class function at any point while an event consumer remains registered.
- Event signals can be warm or hot, as appropriate.
- Event generators and event consumers are de-coupled. In particular this allows for new event consumers⁸ to be registered easily which makes your software more adaptable for future changes as well as making testing easier by enabling patterns such as Mock Objects.⁹

Negatives

- Components are limited to sending event signals between objects within a single thread.
- The pattern doesn't provide any way for security checks to be performed on either the event generators or the event consumers.
- Event generators can be broken by badly behaved event consumers taking a long time to respond to the synchronous event signal.
- When used to define an exported Event Mixin API used across a DLL boundary, it cannot be extended without risking binary compatibility breaks for the event consumers that inherit from the event mixin class.¹⁰
- Event generators need to be aware that event consumers may make some callback into the event generator object whilst it is handling the event signal, i.e. within `MemEventHasOccurredL()`. This can cause problems because the event generator could be in some unstable intermediate state or (potentially more disastrous!) could simply delete the event generator object entirely.

Example Resolved

As mentioned in the problem example, a way is required to send event signals between the generic OBEX protocol implementation, as the event generator, and the service-specific logic, or OBEX server application, as the event consumer. This is solved by creating an event mixin class, `MOBEXServerNotify`, to interface between them.

This event mixin class, defined in `epoc32\include\obexserver.h`, contains several member functions which act as event

⁸Event generators could also be changed easily if a management class was responsible for joining event consumers to event generators.

⁹en.wikipedia.org/wiki/Mock_object.

¹⁰This is due to the way in which vtables are used to enable polymorphism. Adding, removing or reordering virtual functions in a class risks two executables 'disagreeing' about the number or order of virtual functions in a class, which can lead to hard-to-find binary compatibility breaks.

signals for the different events that can happen during an OBEX session with a select few shown below:

```
class MObexServerNotify
{
public:
    virtual TInt ObexConnectIndication(const TObexConnectInfo& aRemoteInfo,
                                       const TDesC8& aInfo) = 0;
    virtual CObexBufObject* PutRequestIndication() = 0;
    ...
};
```

You'll note from the absence of trailing Ls that *Escalate Errors* (see page 32) is not being used to escalate errors from these functions, which shows that alternative error-handling strategies can be used with this pattern. In this case, function return codes are used.

The `ObexConnectIndication()` function is called when a new OBEX session has been established. A `TObexConnectInfo` object is passed into the function, which holds contextual information about the newly connected OBEX client. An OBEX server application inheriting `MObexServerNotify` could therefore implement this function to prepare itself for receiving OBEX requests.

One such request is the Put request to transfer an object from the OBEX client to the OBEX server. When the generic OBEX server code receives such a request, it signals this event to the event consumer by calling the `PutRequestIndication()` function. The OBEX server application is expected to return a pointer from the function to the event generator, either of `NULL` (to indicate that the OBEX server application is unwilling to accept the Put request) or to a `CObexBufObject` which the generic OBEX protocol implementation can use as a receptacle for the object that the OBEX client is sending.

The generic OBEX server protocol implementation, as represented by the `CObexServer` class, has a registration function by which the OBEX server application can provide an `MObexServerNotify*`. Because `CObexServer` can't perform any meaningful OBEX processing without an implementation of `MObexServerNotify`, the semantics of the function are to 'start' the `CObexServer`.¹¹

```
TInt Start(MObexServerNotify* aOwner);
```

In practice, the `MObexServerNotify` class offers an example of how restrictive the event mixin pattern is when used as an API used across a DLL that needs to remain compatible. The set of functions within `MObexServerNotify` cannot be added to without breaking compatibility. In brief, to do so would risk a mismatch between the vtable

¹¹This can also be found in `epoc32\include\obexserver.h`.

entries expected by the DLL that contains the `COBexServer` class and the actual vtable entries present for an `MOBexServerNotify`-derived class contained in some OBEX server application executable compiled against an earlier version of `MOBexServerNotify`.

Other Known Uses

Uses of event mixin classes are numerous in Symbian OS. Some examples include:

- *Messaging Engine*
The `MMsvSessionObserver` class defined in `epoc32\include\msvapi.h` is used by clients of the messaging server to register for notification of messaging-related events. The client of the messaging server, as an event consumer, inherits from this event mixin class and implements the pure virtual function it provides.
Event consumers can then register for messaging-related event signals by using the `CMsvSession::OpenSyncL()` or `CMsvSession::OpenAsyncL()` functions. Having done this, it receives event signals when interesting messaging-related events occur, such as when a new message entry appears in the message index.
- *Eikon*
The `MEikScrollBarObserver` class is inherited by event consumers interested in events related to user interaction with a particular instance of the `CEikScrollBar` class.

Variants and Extensions

- *Registering Multiple Event Consumers with a Single Event Generator*
Here an event generator is not limited to supporting a single event consumer. The event generator may store zero, one or multiple event consumers via event mixin pointers. When dealing with multiple event consumers, it becomes important to ensure that the event consumer de-registration function provided by the event generator takes a reference or pointer to the event consumer to be de-registered, to distinguish it from the other registered event consumers. This variant of the pattern is most similar to Observer [Gamma *et al.*, 1994].
- *Asynchronous Event Mixins*
This pattern is often coupled with *Active Objects* (see page 133) to support asynchronous delivery of event signals. The event generator owns an active object, such as `CAsyncCallback`, which it informs of event signals. This active object completes its own `TRequestStatus` object so that its `RunL()` is executed at the next opportunity to actually send the event signal by synchronously calling the appropriate event

mixin function on the event consumer. Using *Active Objects* (see page 133) helps protect the event generator from badly behaved event consumers by further decoupling them.

References

- Observer [Gamma *et al.*, 1994] is a similar pattern described for a different context.
- *Escalate Errors* (see page 32) describes how to handle error conditions when processing events.
- *Active Objects* (see page 133) is used in conjunction with this pattern if you need an asynchronous event mixin.
- *Publish and Subscribe* (see page 114) is an alternative event-driven programming style that allows event generators to broadcast event signals across thread and process boundaries. It also allows security checks to be applied to both event generators and event consumers.

Request Completion

Intent Use `TRequestStatus` as the basis for event signals sent across thread and process boundaries.

AKA None known

Problem

Context

You wish to propagate warm event signals between an event generator and an event consumer operating within different threads or processes.

Summary

- You wish to reduce the power usage of your component.
- You want to promote encapsulation of and loose coupling between your event generators and event consumers.

Description

An event generator wishes to propagate event signals to an event consumer that is running in a separate thread, possibly in a different process. The fact that a thread boundary is being crossed means *Event Mixin* (see page 93) can't be used to solve the problem.

In addition, the need to support event signals crossing a process boundary means that a number of other possible solutions, such as function pointers, simply won't work. This is because the Symbian OS kernel enforces memory isolation between processes and so an object in one process cannot be directly accessed from another process. In fact, the only way to send event signals between processes is to involve the kernel in some way.

Example

The USB Manager component in Symbian OS uses the transient variant of *Client–Server* (see page 182) to co-ordinate the support for USB. Being a transient server, it is initialized when a client first creates a session to it. When the first session has been created, there is a risk that the client starts to issue requests immediately the session is in place, before the server thread is able to service them, when the server thread is still reading in its initialization data. The client thread must wait until it receives an event

signal to tell it that the server thread is fully initialized and the client can start making requests to it.

Solution

Symbian OS provides low-level, asynchronous, service-handling primitives based around the following two concepts:

- *Asynchronous request status* – this is encapsulated by the `TRequestStatus` class which is essentially a 32-bit value. Many Symbian OS APIs have functions that take a `TRequestStatus&` as a parameter which is an indication that the function provides some asynchronous service. Use of such a function puts the calling code into the role of event consumer and the component that implements the function into the role of event generator. The `TRequestStatus` itself is used by the event consumer to *request* an event signal whilst the event consumer *completes* it to signal that an event has occurred and hence can be viewed as the representation of the event signal.
- *Thread request semaphore* – this is the means by which an event generator tells an event consumer about an event signal. Event consumers can then determine which `TRequestStatus` has completed and hence which event signal it has received. A thread that is waiting for a request to be completed waits on the thread request semaphore (i.e. it does not proceed with any further processing) until the request has been completed and so requires no processor resource, thus saving power until it next needs to do something. The thread does not continue executing until the thread request semaphore is signaled.

Since the completion of a request contains very little contextual information on the nature of the event being signaled,¹² this pattern is best used when implementing higher-level services, such as *Active Objects* (see page 133) or *Client–Server* (see page 182). It is not often used well when applied directly.

An important point to note is that whilst this pattern does work within a thread it is bad practice to use it for intra-thread event signals. This is because better alternatives exist, such as *Event Mixin* (see page 93) and *Active Objects* (see page 133).

Structure

The event consumer (see Figure 4.3) has an instance of `TRequestStatus` (either as a member variable or held on the stack, as appropriate).

¹²That is, a `TRequestStatus` is a weak event signal.

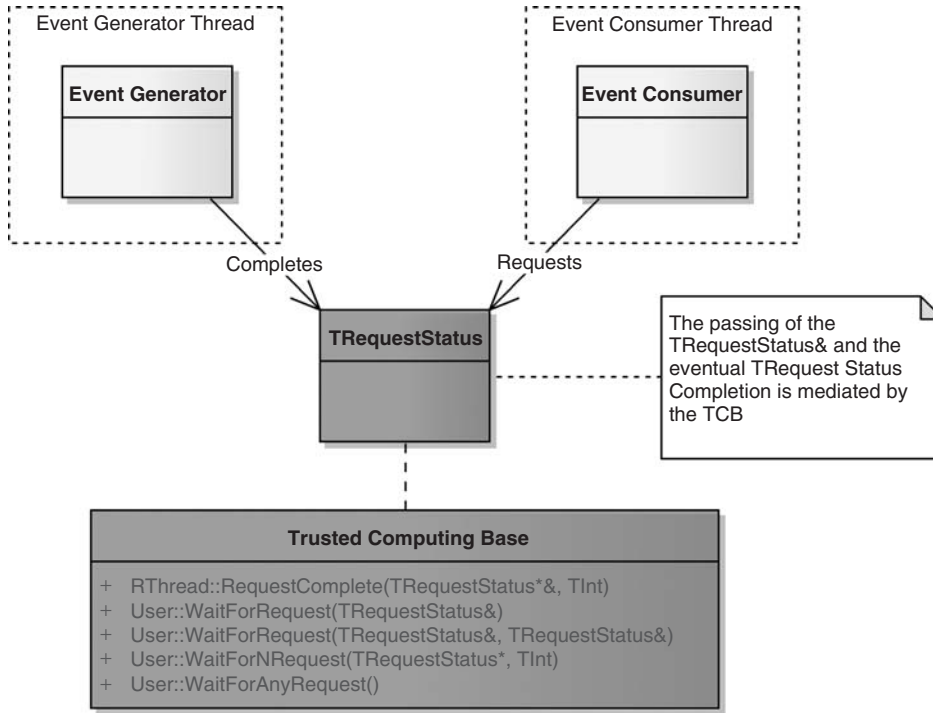


Figure 4.3 Structure of the *Request Completion* pattern

A reference to this `TRequestStatus` is passed to the event generator via the kernel. It then uses one of the `User` functions to wait on the thread request semaphore until one or more event signals have been completed.

The event generator completes the event consumer's request when an appropriate event has occurred by calling the `RequestComplete()` function on the `RThread` object representing the event consumer's thread which signals the appropriate thread request semaphore.¹³

Note that both the event generator and event consumer must be in user threads; they cannot be part of the kernel itself.

Dynamics

Figure 4.4 shows how this pattern operates. Since we are talking about two different threads of execution, it is only really possible to talk about them separately except at synchronization points.

¹³Other functions can be used to complete a `TRequestStatus` but they do so either only within the same thread or only for a specific type of event signal, such as `RProcess::Rendezvous()`, so aren't mentioned here.

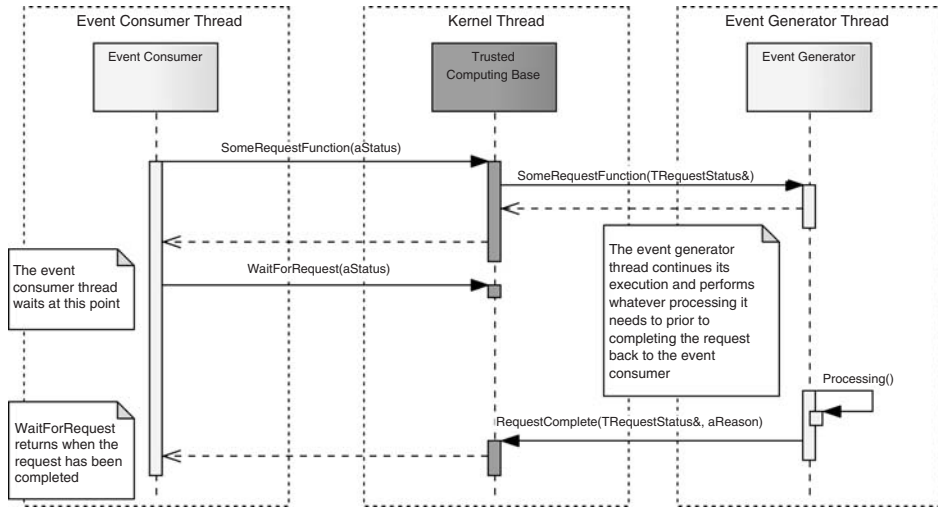


Figure 4.4 Dynamics of the *Request Completion* pattern

Within the Event Consumer Thread

The event consumer passes a reference to its `TRequestStatus` instance to a request function. This function has the responsibility of setting the value of the `TRequestStatus` to `KRequestPending`, to indicate that it has not yet completed, and then passing it to the appropriate kernel service API. The kernel service API synchronously returns to allow the thread to continue execution.

The event consumer may then use one of the `User::WaitForRequest()` functions that take one or more `TRequestStatus&` as a parameter. This causes the thread to stop any further execution until the request has been completed (i.e. this is a synchronization point). It can then continue executing some other task, however, at some point the event consumer thread needs to wait on the `TRequestStatus`.

When the request has been completed the thread continues executing by returning from the `User::WaitForRequest()` function.

Within the Event Generator Thread

The kernel passes to the event generator thread what is effectively a reference¹⁴ to the event consumer's `TRequestStatus`. The event generator does not need to respond immediately and can continue to execute until it is ready to issue the event signal.

This is done by using the function `RThread::RequestComplete()`. This function is called on an instance of `RThread` that acts as a handle

¹⁴It's not a direct reference, to preserve the memory isolation of different processes.

to the event consumer thread. It takes a `TRequestStatus*&` that indicates the request to be completed and a `TInt` which becomes the new value of the `TRequestStatus` in the event consumer thread, and so provides an indication of the outcome of the event. The completion of the `TRequestStatus` causes the event consumer thread to recommence execution.

Implementation

Any given implementation of this pattern generally relies on selecting the appropriate service routine (provided by the kernel) by which the event consumer can pass its `TRequestStatus` to the event generator.

The use of the functions themselves is very simple.

In the Event Consumer Thread

```
TRequestStatus status;  
SomeAsynchronousServiceFunction(status); // Sets status to  
                                           // KRequestPending  
User::WaitForRequest(status); // Thread doesn't continue until status is  
                               // completed
```

In the Event Generator Thread

In the case where the event generator thread receives a `TRequestStatus&` to complete directly, the following code completes the request:

```
TRequestStatus* status = &iStatus;  
consumerThread.RequestComplete(status, result);
```

where:

- `consumerThread` is an `RThread` handle to the event consumer's thread
- `iStatus` is the `TRequestStatus&` to be completed and obtained from the kernel service
- `result` is a `TInt` containing the outcome of the event, such as `KErrNone` for successful completion.

Consequences

Positives

- This pattern is simple to implement.
- The power usage of the components involved is reduced.

- The event generator knows when an event consumer is listening for events.

Negatives

- The kernel must provide a service routine that passes the `TRequestStatus` from the event consumer to the event generator.
- Since the event consumer thread is not executing while waiting for the event, there is no easy way to cancel its interest in the event without the thread being killed outright.
- Only warm event signals can be sent.
- The event consumer must re-register with the event generator following every event signal and hence may miss an event in the time taken to re-register.
- Only a single event consumer per event signal is supported.
- There is no simple way for security checks to be performed on either the event generators or the event consumers.

Example Resolved

The USB example problem can be solved by the use of the kernel thread Rendezvous service. This service allows one thread to wait until another thread signals it has reached the same point; both threads then continue to execute as normal. We can use this to solve our problem as follows:

- Both the USB client thread and the USB Manager thread agree that the rendezvous point they will use is after the client has created the USB Manager and the USB Manager has finished its initializing.
- The USB client thread (the event consumer) calls the `RThread::Rendezvous(TRequestStatus& aStatus)` function passing in a `TRequestStatus`. The client then waits on the `Rendezvous()` until it receives the event signal telling it the USB Manager has also got to the rendezvous point.
- The USB Manager thread (the event generator) calls `RThread::Rendezvous(TInt aReason)` when it has finished initializing. This overload of `Rendezvous()` doesn't take a `TRequestStatus&` because a thread can only be at one rendezvous point at a time and so the kernel doesn't need to be told which `TRequestStatus` it needs to complete.

As you can see, the kernel thread's Rendezvous service is implemented using this pattern. The USB Manager component supplies a client-side

library containing the following code,¹⁵ which executes within the USB client thread. It starts the transient server and uses the kernel thread Rendezvous service to ensure synchronization between it and the new thread:

```
EXPORT_C TInt RUsb::Connect()
{
    TInt retry = 2;

    FOREVER
    {
        TInt err = CreateSession(KUsbServerName, Version(), 10);

        if ((err != KErrNotFound) && (err != KErrServerTerminated))
        {
            return err;
        }
        if (--retry == 0)
        {
            return err;
        }

        err = StartServer();

        if ((err != KErrNone) && (err != KErrAlreadyExists))
        {
            return err;
        }
    }
}

// Start the server process
static TInt StartServer()
{
    const TUidType serverUid(KNullUid, KNullUid, KUsbmanSvrUid);

    RProcess server;
    TInt err = server.Create(KUsbmanImg, KNullDesC, serverUid);
    if (err != KErrNone)
    {
        return err;
    }

    TRequestStatus stat;
    server.Rendezvous(stat);
    if (stat != KRequestPending)
    {
        server.Kill(0);
    }
    else
    {
        server.Resume();
    }
}
```

¹⁵So that it doesn't need to be re-written by each and every client.

```

User::WaitForRequest(stat);

err = (server.ExitType() == EExitPanic) ? KErrServerTerminated :
                                           stat.Int();

server.Close();
return err;
}

```

Here is the server initialization code that executes within the USB Manager thread:

```

GLDEF_C TInt E32Main()
{
    __UHEAP_MARK;

    CTrapCleanup* cleanup = CTrapCleanup::New();

    TInt ret = KErrNoMemory;
    if (cleanup)
    {
        TRAP(ret, RunServerL());
        delete cleanup;
    }

    __UHEAP_MARKEND;
    return ret;
}

// Perform all server initialization, in particular creation of the
// scheduler and server, and then run the scheduler
static void RunServerL()
{
    {
        // Various USB-specific initialization tasks here
        ...

        // Initialization complete, now signal the client
        RProcess::Rendezvous(KErrNone);

        // Ready to run
        // Server activities begin here
        ...
    }
}

```

Other Known Uses

TRequestStatus and its associated functions are ubiquitous within Symbian OS. They form the basis of the following event-driven programming patterns:

- *Active Objects* (see page 133)
- *Client–Server* (see page 182)
- *Publish and Subscribe* (see page 114).

Variants and Extensions

- *Using Multiple TRequestStatus Objects at Once*
The `User` class offers overloads of `WaitForRequest()` which can be provided with references to multiple separate `TRequestStatus` instances. These functions operate in much the same way as the single `TRequestStatus` version, but wait until any of the `TRequestStatus` instances are completed. When the thread continues executing, it must check to see which of the `TRequestStatus` instances were completed and take action as appropriate. This is often useful where you wish to use an asynchronous service that has an upper time limit on completion of the service. In this case, one `TRequestStatus` is passed to the asynchronous service and the other is passed to an instance of `RTimer` or some similar class.
- *The Kernel Is the Event Generator*
A common degenerate form of this pattern is where there is no user-side event generator thread and instead the kernel itself generates event signals. For example, the `RTimer::After()` function allows an event consumer to request the kernel to send it an event signal after a specified time.
- *The Kernel Doesn't Pass on the TRequestStatus*
Here the kernel receives the `TRequestStatus` from an event consumer but doesn't pass it on to the event generator directly. Instead, the event generator uses a kernel service function to inform the kernel of the event signal. The context of the call, or the specific function used, allows the kernel to match this up with the correct `TRequestStatus` and complete it on behalf of the event generator. An example of this is the `RThread::Rendezvous()` function.

References

- Asynchronous Completion Token [Schmidt, 1999] is a similar pattern that allows applications to associate state with the completion of asynchronous operations.
- *Event Mixin* (see page 93) is an alternative to this pattern for when you need to send event signals synchronously within a single thread.
- *Active Objects* (see page 133) is an alternative to this pattern for when you need to send event signals asynchronously within a single thread.

- *Publish and Subscribe* (see page 114) is an alternative to this pattern for when you need to send event signals asynchronously across thread and process boundaries.
- *Client–Server* (see page 182) is a more heavyweight alternative to this pattern for when you need to send event signals synchronously or asynchronously across thread and process boundaries.

Publish and Subscribe

Intent Securely broadcast changes to other threads or processes listening for such notifications as a way of saving power and decoupling components.

AKA P&S

Problem

Context

An event generator needs to broadcast the latest content of a data structure to zero or more event consumers located in other threads or processes that do not need to track each intermediate state of the data structure.

Summary

- You wish to reduce the power usage of your component.
- You want to promote encapsulation of and loose coupling between your event generators and event consumers.
- You would like to perform security checks on either the event generators or the event consumers so that you can control who sends and receives event signals.

Description

The problem that we wish to overcome here is an interesting one because of the large amount of flexibility that is required. First of all, there needs to be no restriction on the number of event generators or event consumers. Secondly, the event generators and the event consumers can be anywhere in the system: in different threads or processes or even in the kernel itself. Clearly *Event Mixin* (see page 93) can't be used to solve this problem by itself!

A natural consequence of this flexibility is that it is not easy to guarantee that each event consumer receives each and every event signal, which is known as *Reliable event signals*. The only completely reliable way to do this is for the event generator to synchronously wait at the point it sends an event signal until every event consumer has received the event signal. Only then should the event generator continue executing. If the event generator doesn't wait, it might find that it needs to send another event signal. So now it's got two to send. Without a synchronization point, the event generator could simply keep generating event signals and exhaust any queue of event signals.

Reliable event signals can be dealt with in two ways:

- Don't even try to support them (the approach taken here). Whilst this restricts the scope of the problem we're addressing, it has the advantage of being significantly simpler to deal with. However, event consumers must only need to access the latest value of the event signal and must not need to know about every intermediate value.
- Support them by generating them one at a time (not discussed here). However, *Client–Server* (see page 182) and *Coordinator* (see page 211) can be combined to provide a solution to this problem.

Example

USB Mass Storage is a technology that provides a standard interface to a variety of storage devices such as portable flash memory devices and digital cameras. A full implementation of this technology requires the Symbian OS File Server to support mass storage drives and an application that allows the end user to mount or un-mount them.

A key requirement of the Mass Storage Application (MSA) is that it should keep the end user informed about the status of the mass storage drives, for example by showing an icon for each drive that is grayed out when the drive isn't present and fully drawn when the drive is present. Rather than polling the File System for this information, the application should register for notification of changes in the mass storage drive status.

This example meets the requirements of this pattern because:

- The MSA is in a different process from the File Server.
- The MSA only needs to show the latest status of each drive to the user since missing an event signal will not inconvenience the end user so long as the status icon is showing the correct state.
- Only the File Server is a valid generator of mass storage events so all other potential event generators should not be allowed to publish an event signal. Conversely, there is no risk in anyone reading the drive status so there is no need to restrict the event consumers.

Solution

This pattern uses the kernel service `RProperty`¹⁶ to allow software to define *properties*. These properties are effectively data structures that can be accessed by components running in any thread or process.¹⁷ One

¹⁶Implemented using *Request Completion* (see page 104).

¹⁷Strictly speaking, this should be 'any process with sufficient platform security capabilities', but more on this later.

notable aspect of properties is that components can register themselves to receive notification when the property value changes, which is known as *subscribing*. In addition, because writing a value into such a property triggers a notification to all subscribers, this is known as *publishing*.

Structure

Defining a Property

A property can be defined by either an event generator or an event consumer (see Figure 4.5) depending on their relative lifetimes. A unique combination of a *category* UID and an integer *key* must be used to identify the property across the system. In addition, the type of the property (such as whether the property value is an integer, Unicode text, a byte array, etc.) must also be set. Once defined, the property value can change, but the property type cannot.¹⁸ Once defined, a property persists in the kernel until the system reboots or the property is explicitly deleted. Its lifetime is not tied to that of the thread or process that originally defined it.

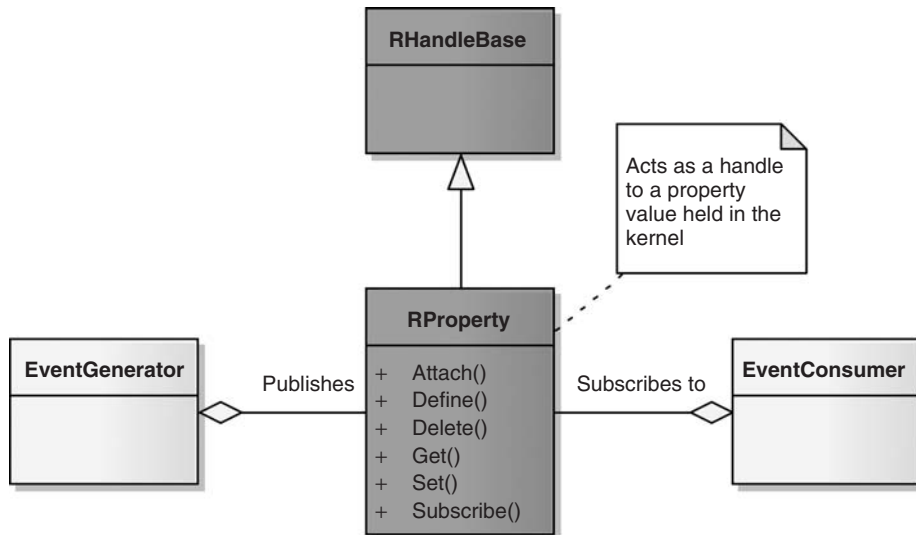


Figure 4.5 Structure of the *Publish and Subscribe* pattern

The `RProperty` API allows byte-array and Unicode text type properties to be pre-allocated when they are defined. This means that the time taken to `Set()` these values is bounded. However, if the length of these property types subsequently increases, then memory allocation may

¹⁸For some property types, the value can change length provided it does not exceed the maximum value of 512 bytes. This puts a limit on the RAM usage in the kernel due to properties.

take place and no guarantees are made about the time taken to `Set()` them.

When a property is defined you have the opportunity to define the policy for the security checks made when it is either written to, by event generators, or read from, by event consumers. By using the `_LIT_SECURITY_POLICY` macros you can specify the Platform Security capabilities, Secure ID or Vendor ID that a process needs to have to succeed either when writing to or reading from the property.

However, there is still a potential risk that some malicious process runs and defines the property with no security policy before you get a chance to define the property correctly yourself. If this is a problem, you should use the special category `KUidSystemCategory`. This works by checking that the process defining such a property has the `WriteDeviceData` Platform Security capability. If the secure definition of a property isn't a worry then you should just use the Secure ID of your process as the category for the property since you know this is unique across the device.

Note that a process that defines a property does not have automatic rights of access to that property, other than to delete it. If the defining process also wishes to publish or subscribe to that property, then it must ensure that it satisfies the security policies that it has put in place when defining the property.

For more information on Platform Security, see [Heath, 2006].

Deleting a Property

Only the process that created a property is allowed to delete it. When you delete a property, any pending subscriptions for it are completed with `KErrNotFound` and any new subscription will not complete until the property is defined and published again. Any `Get()` or `Set()` attempts on a deleted property fail with `KErrNotFound`.

Dynamics

Once a property has been defined it can be used in earnest.

Subscribing

An event consumer can subscribe to a property to request notification of changes in a property's value (see Figure 4.6). This is achieved by the subscriber carrying out the following steps:

1. Create an `RProperty` object.
2. Attach the `RProperty` object to a specific property identified by its category and key.
3. Call `Subscribe()` on the `RProperty` object, passing in a `TRequestStatus&` in the standard manner for an asynchronous service

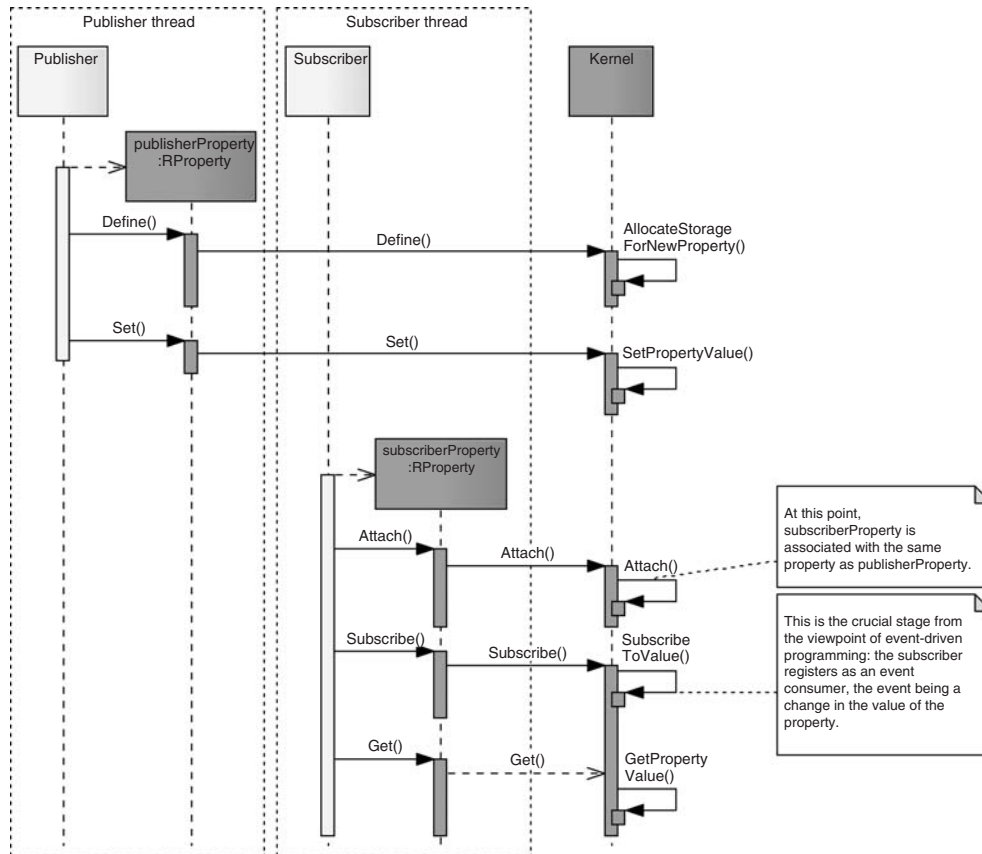


Figure 4.6 Dynamics of the *Publish and Subscribe* pattern (registering a subscriber)

function¹⁹ to register for notification of the next change to the property.

- To actually obtain the current value, you should call one of the `Get()` overloads depending on the type of the property. This operation is atomic, which means that it is not possible for your thread to get a garbled value as a result of another thread writing a value part way through your read. Also, since you have already attached to the property, a `Get()` is guaranteed to have bounded execution time, suitable for high-priority, real-time tasks.

Note that by calling `Subscribe()` before `Get()` there is no opportunity for your component to fail to notice that a new value has been published. In such a situation, when your component calls `Subscribe()` and waits for it to complete, it'll find that it completes

¹⁹See *Request Completion* (on page 104).

immediately and `Get()` is called straight away to obtain the new value.

5. When the property value next changes, the `TRequestStatus` passed into the earlier `Subscribe()` call is completed.²⁰
6. Go to (3) to subscribe for the next change.

Publishing

An event generator can publish a property by doing the following (see Figure 4.7):

1. Create an `RProperty` object.
2. Optionally attach the `RProperty` object to a specific property identified by its category and key. This is only needed if you wish

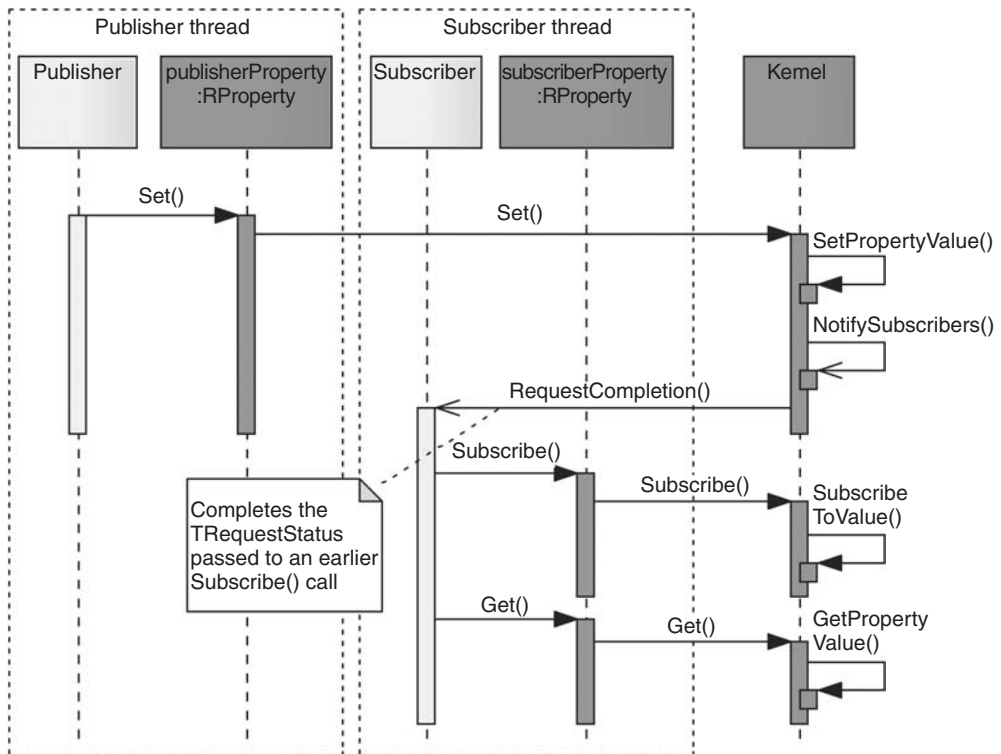


Figure 4.7 Dynamics of the *Publish and Subscribe* pattern (publishing a new value)

²⁰If the event consumer doesn't have sufficient privileges to read the property, the request is completed immediately with `KErrPermissionDenied`.

the subsequent `Set()` operation to be guaranteed to have bounded execution time,²¹ suitable for high-priority, real-time tasks.

3. Publish the new value by calling one of the `Set()` overloads depending on the type of the property.²² An event signal is then sent to all event consumers subscribing to the property.

Once the property has been finished with and is no longer needed, it should be deleted by the process that created it, to free up the RAM it uses in the kernel.

Implementation

Shared Definitions

This pattern works by the event generators and event consumers accessing the same underlying property in the kernel. As such there are some types that need to be shared between them. At the least, this includes the category and key of the property along with some indication of the Platform Security capabilities required to read from it or write to it.

The following code is usually defined in its own header file that is shared between the event consumer and event generator.

```
// Some description should be given here about how this category is used
const TUuid KSharedPropertyCategory = {KProcessSid};

// Some description should be given for each of these enums about how
// the key is used, in particular the type of the property value
enum TSharedPropertyKey
{
    ESharedPropertyKey1,
    ESharedPropertyKey2,
    ...
};

// Example showing how to set a security policy to allow anyone to read
_LIT_SECURITY_POLICY_PASS(KSharedPropertyReadPolicy);

// Example showing how to set a security policy allowing only the
// process that defines the property to write to it
_LIT_SECURITY_POLICY_S0(KSharedPropertyWritePolicy,
    KSharedPropertyCategory.iUid);
```

If the property value's meaning is non-trivial to interpret, you should also provide some means to decode the contents to avoid misinterpretation. For example, if the property is an eight-bit descriptor type containing

²¹It is not guaranteed when publishing a variable-length property that requires the allocation of a larger space for the new value.

²²This operation is atomic, which means that it is not possible for your thread to write a garbled value as a result of another thread attempting to write at the same time.

a packaged class `TExample`, a predefined `TPckgBuf` typedef can easily be provided as follows;

```
typedef TPckgBuf<TPropertyValue> TPropertyValuePckg;
```

Using this kind of typedef to both encode and decode the `TExample` ensures consistent usage of the value for both the publisher and subscriber.

Event Generator

A common practice is to encapsulate the publishing of the new property value within an `R` class.²³ An `R` class is used because this class doesn't own any data and just contains references. This class is expected to be open for the entire time that you wish to use the property and only closed when you have completely finished with the property.

```
class RPropertyPublisher
{
public:
    void OpenL(TPropertyValue& aValue);
    void Close();
    void PublishL();
private:
    RProperty iProperty;
    TPropertyValue& iValue;
};
```

Note that we have chosen to use *Escalate Errors* (see page 32) to handle any errors that occur but you are free to choose a different strategy.

The following function has the responsibility of preparing to publish new values of the property. The property is defined here so that the property only exists whilst this class is Open though for your specific implementation you might find that some notional manager class is better placed to actually manage the lifetime of the property separately from the lifetime of this class.

```
RPropertyPublisher::OpenL(TPropertyValue& aValue)
{
    iValue = aValue;

    // NB Define() is a static function
    TInt result = RProperty.Define(KSharedPropertyCategory,
```

²³Assuming that the `TPropertyValue` object being referenced has a longer lifetime than `RPropertyPublisher`, of course.

```

        ESharedPropertyKey1,
        RProperty::EByteArray,
        KSharedPropertyReadPolicy,
        KSharedPropertyWritePolicy,
        sizeof(TPropertyValuePkg));

    // There's no need to leave if it has already been defined
    if (result != KErrAlreadyExists && result != KErrNone)
    {
        User::Leave(result);
    }

    // Whilst this is optional it does make subsequent publishing faster
    User::LeaveIfError(iProperty.Attach(KSharedPropertyCategory,
        ESharedPropertyKey1));
}

```

The following function has the responsibility of cleaning up the property. In particular, it deletes the property so that we don't unnecessarily waste RAM for a property that isn't required any more. The property is deleted here so that we maintain the class invariant: the property is only valid whilst the class is open.

```

RPropertyPublisher::Close()
{
    iProperty.Close();

    // NB Delete() is a static function
    RProperty::Delete(KSharedPropertyCategory, ESharedPropertyKey1);
}

```

The following function simply has to package up a copy of the new value and publish it via its handle to the underlying property. The event consumer calls this function each time that it wishes to publish the value.

```

void RPropertyPublisher::PublishL()
{
    // Prepare packaged copy of the referenced value
    TPropertyValuePkg valuePkg;
    valuePkg() = iValue;

    User::LeaveIfError(iProperty.Set(valuePkg));
}

```

Event Consumer

The following class encapsulates the subscription to changes in the property. This class derives from `CActive` so that the event consumer thread doesn't have to stop executing whilst it is waiting for an event signal. For more details see *Active Objects* (page 133).

This class is not responsible for handling the event signal; instead it uses *Event Mixin* (see page 93) to allow another class within the event consumer thread to take responsibility for this. Not only does this make the implementation easier to understand here but it is an example of an event-driven chain.

```
class MEventMixin
{
public:
    void HandlePropertyChangeL(TPropertyValue aNewValue);
};
```

The following code is usually defined in a separate header file to the above to allow the packaging also to be decoupled.

```
class CPropertySubscriber : public CActive
{
public:
    static CPropertySubscriber* NewL(MEventMixin* aCallback);
    ~CPropertySubscriber();

protected:
    void ConstructL();

    // From CActive
    void DoCancel();
    void RunL();

private:
    CPropertySubscriber(MEventMixin* aCallback);

private:
    RProperty iProperty;
    MEventMixin* iEventMixin;
};
```

For brevity, we've not shown the implementation of the constructor which simply stores the callback passed in as a parameter and sets up the active object appropriately. The `NewL()` is no different from a normal two-phase constructor implementation. However, the `ConstructL()` is responsible for preparing its `RProperty` object for later use and starting the calling `Subscribe()` for the first time via the `RunL()`:

[illegible]

The destructor is simply responsible for clearing up its resources:

```
CPropertySubscriber::~CPropertySubscriber()
{
    Cancel();
    iProperty.Close();
}
```

The `Cancel()` above calls the following function if a `Subscribe()` call is outstanding:

```
void CPropertySubscriber::DoCancel()
{
    iProperty.Cancel();
}
```

Here is where the majority of the work is done, following the standard publish and subscribe pattern of `Subscribe()` and `Get()` followed by the handling of the event signal:

```
void CPropertySubscriber::RunL()
{
    User::LeaveIfError(iStatus);

    // Register for further event signals before handling the new value
    iProperty.Subscribe(iStatus);
    SetActive();

    // Get the new value
    TPropertyValuePkg valuePkg;
    User::LeaveIfError(iProperty.Get(valuePkg));

    // Send event signal on the event mixin to handle the value
    iEventMixin->HandlePropertyChangeL(valuePkg());
}
```

Consequences

Positives

- This pattern reduces the power usage of your components.
- The publish and subscribe service provided by the kernel is simple to use.
- It offers a way to communicate values and event signals between event generators and event consumers in different threads or processes.
- Event generators and event consumers are very loosely coupled with few static dependencies between them. The only ones should be captured in your `sharedproperty.h` header file.

- It allows security checks to be performed separately on both event consumers and event generators.

Negatives

- It doesn't support reliable event signals and hence can't be used in situations where an event consumer needs to be aware of every intermediate property value.
- It only supports warm event signals since the actual value of the property must be retrieved via `RProperty::Get()`.
- Event generators have no knowledge about whether there actually are any event consumers receiving the event signals.

Example Resolved

To resolve the mass storage example, the File Server, as the event generator, takes the responsibility of defining a property through which the state of the mass storage drives are published to event consumers such as the MSA. The MSA application simply subscribes to these events and updates its UI as appropriate.

Shared Definitions

The following types are defined in `usbmsshared.h`: the shared category and four properties, including the one used by the MSA application to monitor the status of mass storage drives – `EUsbMsDriveState_DriveStatus`.

```
// The publish and subscribe category for all USB Mass Storage events.
const TUid KUsbMsDriveState_Category = { KFileServerUidValue };

// The publish and subscribe event key enumeration
enum TUsbMsDriveState_Subkey
{
    EUsbMsDriveState_DriveStatus,
    EUsbMsDriveState_KBytesRead,
    EUsbMsDriveState_KBytesWritten,
    EUsbMsDriveState_MediaError
};
```

In addition, `usbmsshared.h` defines a data structure, `TUsbMsDrivesStatus`, that contains information on up to eight drives giving the drive number and the state it is in:

```
// Possible values for each of EUsbMsDriveState_DriveStatus status
// codes
enum EUsbMsDriveStates
```

```

{
    /** File system not available for Mass Storage */
    EUsbMsDriveState_Disconnected    =0x0,
    /** Host has required connection */
    EUsbMsDriveState_Connecting      =0x1,
    /** File system available to Mass Storage */
    EUsbMsDriveState_Connected       =0x2,
    /** Disconnecting from Mass Storage */
    EUsbMsDriveState_Disconnecting   =0x3,
    /** Critical write - do not remove card */
    EUsbMsDriveState_Active           =0x4,
    /** Connected, but locked with a password */
    EUsbMsDriveState_Locked           =0x5,
    /** Connected, but card not present */
    EUsbMsDriveState_MediaNotPresent =0x6,
    /** Card removed while active */
    EUsbMsDriveState_Removed          =0x7,
    /** General error */
    EUsbMsDriveState_Error            =0x8
};

// The maximum number of removable drives supported by Mass Storage
const TUint KUsbMsMaxDrives = 8;

// A buffer to receive the EUsbMsDriveState_DriveStatus property
// For each drive there is a pair of bytes: the drive number
// followed by the EUsbMsDriveStates status code.
typedef TBuf8<2*KUsbMsMaxDrives> TUsbMsDrivesStatus;

```

Event Generator

The File Server uses a class called `RDriveStateChangedPublisher` to publish the drive status information. This is done in a very similar way to the implementation given above, in that the class constructor takes references to the information to be published so that the user of the class, `CDriveManager`, simply has to ask it to publish the new value.

One difference from the standard pattern implementation above is that the security policies to read and write the `EUsbMsDriveState_DriveStatus` property are not defined in the shared class because it's not strictly necessary. Instead, they're defined locally within the `RDriveStateChangedPublisher` implementation as:

```

_LIT_SECURITY_POLICY_PASS(KMassStorageReadPolicy);
_LIT_SECURITY_POLICY_S0(KMassStorageWritePolicy,
    KUsbMsDriveState_Category.iUid);

```

This allows any event consumer to read the property but only the File Server may publish the drive status information as required.

Event Consumer

The MSA application simply subscribes to the `(KUsbMsDriveState_Category, EUsbMsDriveState_DriveStatus)` property

and passes each new value of the property on via the following event mixin class:

```
class MDriveWatch
{
public:
    virtual void DrivePropertyChange(TUsbMsDrivesStatus aDriveStates) = 0;
};
```

The MSA application contains a `CMSManager` class that derives from `MDriveWatch` to handle the fact that the property has changed by unpacking `aDriveStates` and updating each drive icon in its UI appropriately.

Other Known Uses

This pattern is used by many components throughout Symbian OS, for example:

- *Time Zone Server*
Uses this pattern to provide notifications of changes to:
 - System-wide UTC offset changes, via the `EUtcOffsetChange-Notification` key
 - The current time zone, via the `ECurrentTimeZoneId` key
 - The home time zone, via the `EHomeTimeZoneId` key
 - The Next Daylight Saving Time Change, via the `ENextDSTChange` key.

See `epoc32\include\tzupdate.h` for more information on the shared types used.

- *Backup Server*
Uses this pattern to indicate whether a backup is in progress or not via the `(KUidBackupRestoreKey, KUidBackupRestoreKey)` property. See `epoc32\include\connect\sdefs.h` for more information on the shared types used.

Variants and Extensions

- *Many Publishers, Single Subscriber*
Normally an event generator defines a property in order to be able to share a property value with one or more event consumers. Such is the convenience of this pattern that occasionally an event consumer might define a property in order to subscribe to the property itself, defining a suitably permissive write policy for the property such that

one or more other event generators are able to update the property value.

While this is of limited use if the property owner–subscriber must know each value of the property, it can provide a convenient means of allowing external components to trigger some action in the event consumer while remaining very loosely coupled. This enables easy customization of the event consumer since it is not reliant on event generators even being present and reacts immediately once they are installed on the device.

- *Multiple Publishing Frequencies*

When a property is changed, all event consumers are notified. This leads to their threads running to service the event signal. If a property changes value frequently, it is wasteful for event consumers to perform substantial processing for each event signal.

Take a property representing signal strength as an example. Potentially, this could be updated several times a second. If a change in value were used only to update the UI signal bar, it would not be harmful. However, if it were used by many entities for serious processing (e.g. checking email, sending unsent SMS messages, re-connecting to the Internet), then such frequent updates would have a severe effect on battery life.

Nevertheless, it is obviously desirable for many components of a device to know about the state of network coverage and to take appropriate action. In cases like this, it may be worth the event generator defining multiple properties with associated update characteristics. For example, raw signal strength (updated more than once a second), periodic signal strength (updated once every 10 seconds) and network coverage (updated only when moving between some network signal and none). Each event consumer can then monitor the appropriate notification and so reduce the number of threads that run when the underlying value changes.

References

- Publisher–Subscriber [Buschmann *et al.*, 1996] describes a similar pattern for a different context.
- *Event Mixin* (see page 93) is often composed with this pattern to separate the responsibility of subscribing to an event from handling the event signal.
- *Active Objects* (see page 133) is often composed with this pattern to avoid the need to synchronously wait for the event signal to complete.

- *Client–Server* (see page 182) and *Coordinator* (see page 211) can be combined to provide a solution to reliable event signals that this pattern doesn't support.
- *Escalate Errors* (see page 32) is used to handle errors when dealing with the `RProperty` API.
- *Request Completion* (see page 104) provides more details on how `TRequestStatus` is used as an event signal within the *Publish and Subscribe* pattern.
- The following document provides additional information on the publish and subscribe technology: Symbian Developer Library » Symbian OS guide » Base » Using User Library (E32) » Publish and Subscribe.

5

Cooperative Multitasking

Multitasking is where multiple tasks share common resources such as a CPU or data. In the case of a device with a single CPU,¹ only one task is said to be running at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking involves scheduling which task is to be the one running at any given time whilst all other tasks wait to be given a turn. By switching between tasks frequently enough, the illusion of parallelism is achieved. One thing to remember here is that multitasking is a more general concept than multithreading since whilst individual tasks can be assigned to run in a single thread this is not always necessary.

As discussed in Chapter 4, Symbian OS makes extensive use of event-driven programming to conserve power whilst promoting the decoupling of components. This naturally leads to multitasking as any moderately complex software will need to deal with incoming event signals from multiple sources. Early GUI systems took a step backwards from the big systems of the previous era by implementing their event-driven programming models using non-pre-emptive process scheduling (*co-operative multitasking* as it became known). This was in contrast to the UNIX and mainframe approach of pre-emptive process scheduling, in which the currently running process can be swapped out at any time by the operating system scheduler. In a non-pre-emptive system, the scheduler chooses the process which will run next, but always lets it run to completion. Pre-emption incurs more overhead and makes the system more complex. But the non-pre-emptive approach has the serious drawback that badly behaved applications can break a co-operative system by not yielding to allow other processes to be scheduled.

Multithreaded systems introduce finer-grained, lighter-weight thread management, in addition to process management, as an attempt to keep

¹ At the time of writing, all Symbian OS applications run on a single CPU but this is set to change in future with SMP (www.symbian.com/symbianos/smp). However, multitasking will still occur on each individual CPU.

the benefits that multiple process systems bring, including asynchronous and event-driven programming paradigms and fully pre-emptive systems, without suffering their performance penalties. The price we pay for this is increased software complexity. Two problems in particular make writing multithreaded programs difficult and error prone:

- *Synchronization* – for example, to control the execution order of your threads. This is necessary if the sequence in which your threads complete would otherwise be indeterminate.
- *Shared data and the problem of atomicity* – reads and writes from one thread may be invalidated partway through by a pre-empting thread rewriting the data. This requires that these data reads and writes are atomic such that once a thread has begun an operation no other thread can attempt the same operation until the first thread has finished.

So while multithreading is becoming increasingly common in applications programming,² most systems leave the individual developer to solve these problem of complexity. Many systems, including Symbian OS, provide tools (mutexes, semaphores, critical sections, etc.) that help allow you to co-ordinate threads and handle user events but these only make it possible to write multithreaded code rather than solve the problem.

Another aspect to consider is the overhead of each thread on a Symbian OS device with limited RAM. As discussed in Appendix A, a thread uses by default at least 17 KB of RAM.

The ideal situation for Symbian OS is to have pre-emptive scheduling of multiple different processes across the system combined with single-threading of each process to give us cooperative multitasking within a process.

The patterns in this chapter provide a way of doing just that. *Active Objects* (see page 133) describes the basis of the Symbian OS approach to cooperative multitasking within a single thread whilst *Asynchronous Controller* (see page 148) builds on that by describing a technique to organize a series of related sub-tasks to achieve some larger task.

²It used to be the preserve of systems and server programmers.

Active Objects

Intent Enable efficient multitasking within a single thread by using the Symbian OS active object framework.

AKA None known

Problem

Context

Your application or service needs to continue executing other tasks while waiting for the completion of an asynchronous service.

Summary

- You wish to reduce your maintenance costs, such as by reducing the time that you spend debugging your software.
- You wish to reduce the time that you need to spend testing your software looking for defects.
- You wish to reduce the amount of RAM used by your software, for example by reducing the number of threads running in your process.

Description

There are many kinds of services that are naturally asynchronous and which therefore expose asynchronous APIs to clients. In general, asynchronous APIs are likely to be found wherever you have an interface to a component that is unpredictable or irregular; has a potentially high response time;³ or which in any sense displays indeterministic behavior. A good strategy is to treat virtually any event signal coming from outside the current process in an asynchronous manner so that you are less dependent on components outside of your control. Examples of this range from end users generating input of any kind to communication lines that can go down and phones that could get left off the hook.

Any program that deals with event generators, invokes callbacks or shares a resource through *Client–Server* (see page 182) will have to deal with asynchronous APIs. All event-driven programs, including interactive applications, are naturally asynchronous, at least in their event handling.

³Known as *latency*.

Services provide asynchronous APIs to allow them to be decoupled from their clients. The way that this works is that requests are made to the service and are guaranteed to return immediately. At this point, the service has simply registered the request. By returning without processing the request, the client is decoupled from the processing of the service request, which may take an indeterminate time or may fail.

When dealing with a service that handles requests asynchronously you could halt the execution of your thread and wait until the service handles your request and sends you the result. However, this defeats the whole purpose of the service providing an asynchronous API and instead you need to find some way to continue to treat the period of waiting and the subsequent handling of the response as just another task that should be scheduled alongside all the other tasks you need to execute. Hence you need to multitask. The problem is to do so without using a multithreaded solution to avoid the complications and RAM overhead described in the introduction to this chapter.

Example

Applications that wish to make a phone call can do so by accessing the Telephony subsystem of Symbian OS through the `CTelephony` API. This API is asynchronous due to the relatively high latency of establishing a connection.

Of course, the main application thread could block while waiting for the call to be connected but this would render any UI controls unresponsive and would probably frustrate the end user. A better strategy is to adopt a cooperative multitasking approach executed within a single thread.

Solution

The Symbian OS active object framework provides developers with a consistent, uniform, and general-purpose model for managing asynchronous behavior without resorting to multithreading.

An active object is used by some client object to encapsulate the task of making an asynchronous service request and handling the eventual request completion all within a single thread but without blocking until the request completes. This pattern builds on *Request Completion* (see page 104) to provide a more usable, higher-level API that shields developers from the trickiness of multitasking, by moving complexity out of sight into the internals of the operating system.

For many purposes, active objects are as effective as threads. Unless there is an explicit reason not to use them, Symbian OS active objects are usually the simplest solution for managing asynchronicity.

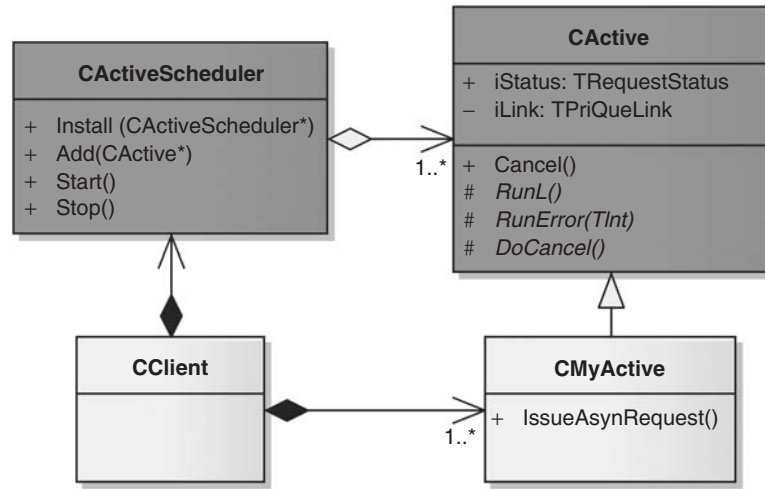


Figure 5.1 Structure of the *Active Objects* pattern

Structure

There are two principle classes in the active object framework⁴ in addition to your concrete active objects (see Figure 5.1).

- Active scheduler represented by CActiveScheduler*

This class is responsible for managing and scheduling a list of active objects within a single thread. Each active scheduler maintains its own list of runnable active objects, and chooses the next object to run based on its active object priority or, when several active objects have the same priority, in First-In–First-Out (FIFO) order.

Note that any code running in the thread can add active objects to the active scheduler which includes code provided by another vendor that is included as a DLL. An example of this is a service designed according to *Client-Thread Service* (see page 171). This can be an issue because it is more difficult to arrange the active objects from different vendors to cooperate well at run time. Since the active scheduler runs a non-pre-emptive scheduling scheme there is no way to recover if a single, high-priority active object prevents all other active objects from running.
- Active object represented by CActive*

This defines the abstract base class from which all concrete active objects are derived. The first thing you need to do is define the asynchronous request to be encapsulated by an active object and provide a function that issues the request. This function would use the `TRequestStatus` it owns and pass it into the service providing

⁴Defined in `epoc32\include\e32base.h`.

the asynchronous API. It is this `TRequestStatus` object that is completed when the request has been completed.⁵

The `CActive` class uses virtual functions and the Template Method pattern [Gamma *et al.*, 1994] to define the responsibilities of concrete active objects. As a minimum, you have to provide an implementation of `RunL()`, to handle the completion of your asynchronous request, and `DoCancel()`, to provide a way to tell the asynchronous service provider to attempt to stop processing your request if it is still outstanding.⁶

If required, you can also implement `RunError()` to provide your own specific error handling since this function is called if a `Leave` occurs within your `RunL()`.

Dynamics

Setting up the Active Scheduler

When a thread is created, an active scheduler must be created and installed for it. Before it can be started, at least one active object with an outstanding request must be added to the active scheduler before it is started. An active scheduler is then normally not stopped until the thread exits. The steps required to set up the active scheduler for a thread with a single active object are shown in Figure 5.2.

Normal Request Completion

Figure 5.3 shows a simple request that completes normally. Within the `RunL()`, the active object should first check to see if its request completed with an error and handle the success or failure of the request appropriately.

At a system-wide level, any running active object can be pre-empted by the kernel's normal thread scheduling mechanism, since an active object is run within a thread like any other executable object. However, it is important to note that, locally, all the active objects run by a given active scheduler multitask cooperatively; once an active object is started running, it runs to completion before the next ready-to-run active object can be started by that scheduler.

You need to be aware of this to ensure that the active objects you create run efficiently as well as to avoid unintended side effects such as one active object blocking others within the thread from running. Hence it is important that the `RunL()` completes quickly, for instance, within 100 ms,⁷ so that your thread remains responsive to further events from the end user or elsewhere.

⁵See *Request Completion* (page 104).

⁶`DoCancel()` is only called by the framework when it knows there is a request outstanding.

⁷As this is roughly the threshold at which a delay is detectable by an end user. See also the Long Running Active Object variation of this pattern.

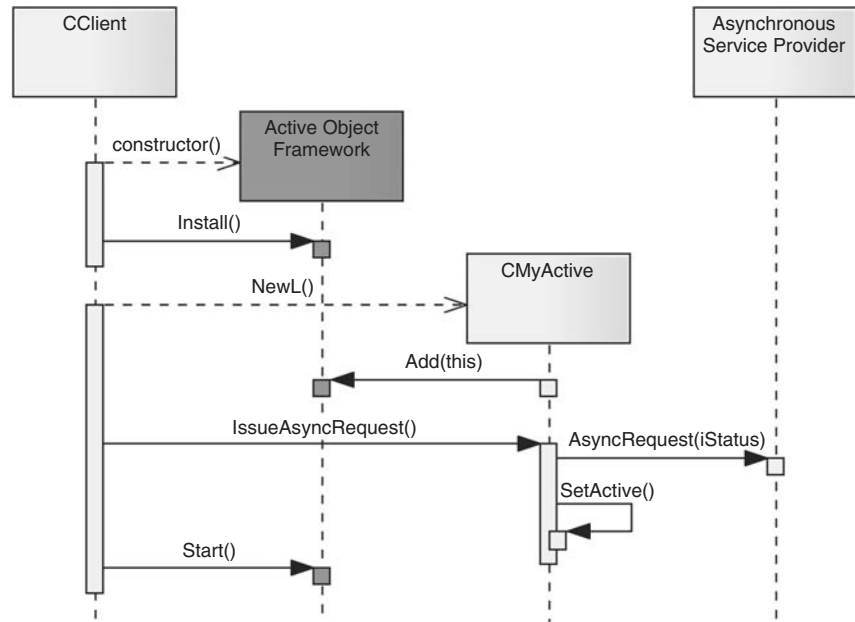


Figure 5.2 Dynamics of the *Active Objects* pattern (setting up the active scheduler)

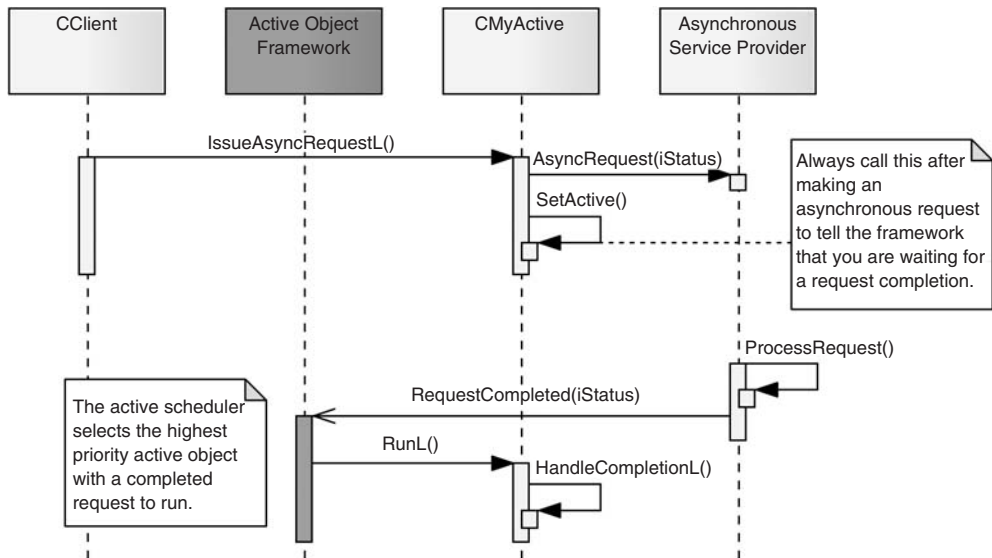


Figure 5.3 Dynamics of the *Active Objects* pattern (normal request completion)

Canceling an Asynchronous Request

The primary purpose of `CActive::Cancel()` is to tell the asynchronous service provider that you have no more interest in the request. As the implementer of an active object, you need to ensure this happens by calling the appropriate `Cancel()` function given by the asynchronous service provider from within your `DoCancel()` function (see Figure 5.4).⁸ The active object framework then blocks until the service provider completes the request. However, this does not result in a `RunL()` being called since you've expressed no interest in the request by calling `Cancel()`. It also avoids any problem with a badly written `RunL()` firing off another request just when you wanted to destroy the active object.

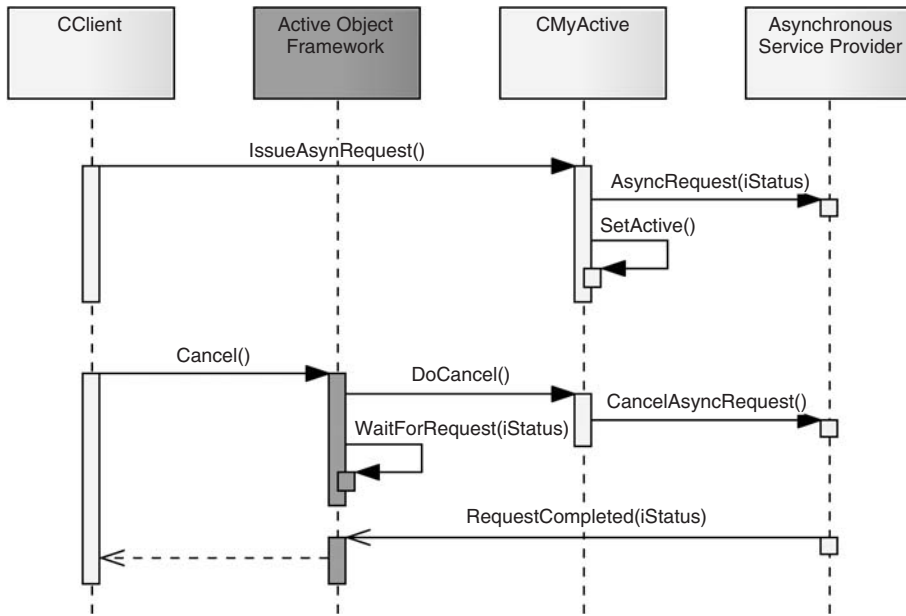


Figure 5.4 Dynamics of the *Active Objects* pattern (canceling an asynchronous request)

Implementation

Active objects are thoroughly documented in the Symbian Developer Library as well as in many books from Symbian Press such as [Stichbury, 2004] and [Harrison and Shackman, 2007]. However, the following summarizes the most important points to remember when creating and using this pattern.

⁸Your `DoCancel()` is only called if there is a request outstanding.

Setting up the Active Scheduler

An active scheduler is normally one of the first things created when a thread runs, often in a local function called directly from the `E32Main()` thread entry point such as the following:

```
static void RunThreadL()
{
    // Create and install the active scheduler
    CActiveScheduler* s = new(ELeave) CActiveScheduler;
    CleanupStack::PushL(s);
    CActiveScheduler::Install(s);

    // Prepare the initial list of active objects
    CMyActive* myActive = CMyActive::NewLC(); // Adds myActive to the
    myActive->IssueAsyncRequest();           // scheduler
    ...

    // Ready to run
    CActiveScheduler::Start(); // Doesn't return until it is explicitly
                               // stopped

    // Clean up the scheduler and myActive
    CleanupStack::PopAndDestroy(2);
}
```

Note that you do not need to do this for applications as the application framework creates an active scheduler for you.

Creating an Active Object

Your active object will look something like this:

```
class CMyActive : public CActive
{
public:
    static CMyActive* NewL();
    ~CMyActive();

    void IssueAsyncRequest();
private:
    CMyActive();

    // from CActive
    void RunL();
    TInt RunError(TInt aError);
    void DoCancel();
};
```

When implementing the active object, your first task in the constructor is to set up the priority used by it. Exactly what value you use depends on the other active objects it is cooperating with. However, the priority should be chosen with care. The list available to

choose from is given by the `CActive::TPriority` enum defined in `epoc32\include\e32base.h`. Remember that an object with a high priority will be chosen to run ahead of others, which means that it is possible for one object to starve others of processor time if it is made active again at the end of its `RunL()`.

You should also register the current object with the active scheduler:

```
CMyActive::CMyActive()
: CActive(CActive::EPriorityStandard)
{
    CActiveScheduler::Add(this);
}
```

When implementing the function that issues the asynchronous service request you should check to see if your object is already active. This check *is* important. If an active object succeeds in making a new request before the currently outstanding request has completed, a *stray signal* will result⁹ and cause a system panic, since the first request won't be able to complete correctly. It is appropriate to panic if the function is called incorrectly as per *Fail Fast* (see page 17) since it indicates a fault in the calling code.

However, if an error occurs which cannot be handled simply by panicking and terminating the thread then you should not `Leave` from the function, as per *Escalate Errors* (see page 32). The reason for this is that you will already have one error path at the point at which the request completion is handled and by leaving here you only introduce another error path which unnecessarily complicates calling code. Instead you should ensure that you re-use the existing error path, perhaps by calling your request completion handling function directly.

Last but not least, don't forget to actually issue the asynchronous request to the asynchronous service provider, accessed through `iService`. Most Symbian OS APIs do this by taking a `TRequestStatus` object to allow the service to pass back the request completion event.¹⁰ It's important that you use the `iStatus` provided by the `CActive` base class so that the completion is handled as described below in the active object's `RunL()` function.

```
void CMyActive::IssueAsyncRequest()
{
    ASSERT(!IsActive());

    // Actually issue the request
    iService->AsyncRequest(iStatus);
}
```

⁹This is when a request has been signaled as complete and either there is no active object available or an active object is found that has neither been set active nor has a request pending. The result is an `E32USER-CBase` panic.

¹⁰As per *Request Completion* (see page 104).

```
// Tell the framework that a request is pending but
// do it after the async request in case it Leaves
SetActive();
}
```

You must implement a `RunL()` where you should handle completion of the asynchronous request issued by `IssueAsyncRequest()` as appropriate to your situation:

```
void CMyActive::RunL()
{
    // As an example, this Leaves if the request completed with an error so
    // that errors are handled in RunError()
    User::LeaveIfError(iStatus);

    // Implementation-specific, request-handling code
    ...
}
```

It is optional to implement the `RunError()` method but if the `RunL()` may Leave and you need error-handling specific to this active object then you should do so and use it to resolve any errors that occurred for your original request. You should return `KErrNone` for errors that are handled within the function but return the actual error code for any errors that are not handled.

If you don't handle an error within your active object, it'll be passed to the active scheduler to resolve. Exactly how this is done depends on what platform you're on.¹¹ However, it is common practice within applications for your `CxxxAppUi::HandleError()` function to be called. If that doesn't handle the error then on UIQ a dialog is shown to the end user whilst on S60 the application is silently terminated.

```
TInt CMyActive::RunError(TInt aError)
{
    // For example we can resolve KErrNotFound errors here ...
    if (aError == KErrNotFound)
    {
        // resolve error such as by retrying
        IssueAsyncRequest();
        return KErrNone; // error resolved
    }

    return aError; // error not handled
}
```

In the destructor of the active object, in addition to all the normal things you'd do, such as clean up any resources you've used, you should always call `CActive::Cancel()` to ensure that there is no outstanding

¹¹Since device manufacturers can customize the active scheduler by deriving from `CActiveScheduler`.

request at the point of destruction. Failing to do so and leaving a request outstanding *will* cause a stray signal when the asynchronous call completes and the active scheduler tries to invoke the `RunL()` of an object that no longer exists. Remember you should never call `DoCancel()` directly because, unlike `Cancel()`, this doesn't check if there is an outstanding request before sending a cancel to the service provider.

```
CMyActive::~CMyActive()
{
    Cancel();

    // Other cleanup code
    ...
}
```

You must implement the `DoCancel()` method to call the cancel method that matches the asynchronous request made in `IssueAsyncRequest()`:

```
void CMyActive::DoCancel()
{
    iService->CancelAsyncRequest();
}
```

Finally, it is a common mistake for developers to declare an unnecessary `TRequestStatus` member variable in their active object. Don't do this – instead use the one supplied by the `CActive` base class. Nor is it necessary to set the `iStatus` value to `KRequestPending` after making a call; the framework sets this value in `SetActive()`.

Consequences

Positives

- Especially for applications, where the framework takes care of managing the active scheduler, active objects are simple to create and use.
- Active objects are significantly more lightweight than threads, with much less RAM overhead. For example, in Symbian OS the absolute minimum overhead for a new thread is 9 KB of RAM, approximately 20 times the overhead of introducing an active object where the main cost is the additional code size of roughly 500 bytes.¹²
- In most cases, active objects are as effective as multithreading but are easier to debug, which reduces your maintenance and testing costs. This is not say debugging them is always easy, as stray signals

¹²The average code size for a class within Symbian OS itself.

can be a pain to track down, however the problems introduced by multithreading are usually more complex and hard to identify.

- The active object priorities provide you with a flexible mechanism for ensuring that the important events, such as handling input from the end user, are dealt with before low-priority tasks such as clearing out a cache.
- Switching between active objects running in the same thread (which is the typical use case) avoids the overhead of a thread context switch, so a component that uses this pattern, instead of threads, to manage multiple tasks is normally more efficient.

Negatives

- Since active objects are cooperatively scheduled, this introduces dependencies between apparently independent parts of a component, so that badly written active objects in one place can cause trouble elsewhere. Any code, such as that provided in a separate DLL, can create an active object and effectively disrupt the entire thread.¹³
- Since this pattern schedules tasks cooperatively, it is not ideal for situations where real-time responses are required. This is not to say it won't work, just that it'll take some care to set up and especially to maintain so that alternative scheduling techniques such as pre-emptive multitasking would probably be more appropriate.
- The active scheduler uses a single list to manage its active objects and is not intended to support a very large number of active objects. Doing so will impact the performance of your thread since the active scheduler must linearly traverse the list of all active objects that it manages in order to find the correct `TRequestStatus` to complete, which won't scale well.
- Since this pattern is not directly supported by the standard C++ libraries, it is less suitable when porting existing C++ code to or from Symbian OS.

Example Resolved

The example given above was an application wishing to make a phone call. Since this is an asynchronous task, this pattern can be used to encapsulate it within an active object as follows:

```
#include <e32base.h>
#include <Etel3rdParty.h>
```

¹³If you're worried that someone might do this deliberately then you shouldn't load the DLL into your process. See Chapter 7 for alternative ways of tackling this problem.

```

class CDialer : public CActive
{
public:
    CDialer(CTelephony& aTelephony);
    ~CDialer;

    // Equivalent of IssueAsyncRequest()
    void Dial(CTelephony::TTelNumber aNumber);

private: // From CActive
    void RunL();
    void DoCancel();
private:
    CTelephony& iTelephony; // Service provider
    CTelephony::TCallId iCallId;
};

```

In this case, we don't need to do any specific error handling in this object, so no `RunError()` is provided.

The constructor is fairly normal since the priority given to the object is `EPriorityStandard`:

```

CDialer::CDialer(CTelephony& aTelephony)
: CActive(EPriorityStandard), iTelephony(aTelephony)
{
    CActiveScheduler::Add(this);
}

```

The `Dial()` function is where we start the whole process of making a phone call using `CTelephony::DialNewCall()`. In particular, you should note that the `TRequestStatus` of this object is being passed in, which shows this is an asynchronous call and is how the active scheduler knows which active object initialized this request.

In addition, the API takes an output parameter as an argument – `iCallId`. This is an object of type `CTelephony::TCallId` and is a token used to identify the call that you can use if you need to perform further operations, such as putting the call on hold or hanging up. However, this will not be a valid object *until the request has completed* so it shouldn't be accessed until the `RunL()` fires. It is also significant that the object is owned by `CDialer` rather than just being constructed on the stack, as `callParams` is below. This ensures that the object is still in scope by the time the `RunL()` is called later. This can be quite a subtle bug if you get it wrong because often the service provider has a higher priority than its clients. Hence in normal operation the service request will cause a context switch to the service provider's thread which handles and completes the request so that when control is returned to

your thread, the object on the stack is still in scope. However, this is not guaranteed even now due to *priority inversion*¹⁴ and will be more of a problem for SMP¹⁵ devices in the future.

```
void CDialer::Dial(CTelephony::TTelNumber aNumber)
{
    CTelephony::TCallParamsV1 callParams;
    callParams.iIdRestrict = CTelephony::ESendMyId;
    CTelephony::TCallParamsV1Pkg callParamsPkg(callParams);

    iTelephony.DialNewCall(iStatus, callParamsPkg, aNumber, iCallId);
    SetActive();
}
```

In the following function, the request to dial a new call needs to be handled:

```
void CDialer::RunL()
{
    User::LeaveIfError(iStatus); // Rely on the default error handling

    // The call has been dialled successfully and
    // iCallId now contains the call's ID

    // Update the UI (not shown for brevity)
    ...
}
```

We also need to provide a way to cancel the dial request:

```
void CDialer::DoCancel()
{
    // Note that this doesn't mean "do the cancel asynchronously" but
    // rather means "cancel the asynchronous request made earlier"
    iTelephony.CancelAsync(CTelephony::EDialNewCallCancel);
}
```

Finally, the destructor simply needs to cancel any outstanding requests since it doesn't own any of its own resources:

```
CDialer::~CDialer()
{
    Cancel();
}
```

¹⁴See en.wikipedia.org/wiki/Priority_inversion.

¹⁵See www.symbian.com/symbianos/smp.

Other Known Uses

Active objects are used extensively throughout Symbian OS, so we give just two examples here:

- *Servers*
This pattern is used extensively within *Client–Server* (see page 182) with every server object implemented as an active object. In addition, servers often use additional active objects, called servants, to process client requests that they can't satisfy immediately so that they continue to remain responsive to new client requests.
- *Window Server*
This pattern forms the basis of the Window Server's event-handling framework, which handles all application events such as key presses, pointer events, animations, etc.

Variants and Extensions

- *Single Use Active Object*
This pattern describes an active object as a manager of multiple asynchronous requests on behalf of some client. However, a slightly simpler form of this pattern is when the active object only exists whilst a single asynchronous request is outstanding. By making this design choice, you can simplify the active object and its use by the client as you are then able to implement the active object so that it makes the asynchronous request when it is constructed. The client then doesn't need to separately remember to tell the active object to start the asynchronous request and nor does it need to worry about whether the active object has a request outstanding; if it exists then a request is pending. To maintain this design, once a request has been completed the client must ensure it deletes the associated active object.
- *Asynchronous Event Mixins*
This pattern is often coupled with *Event Mixin* (see page 93) to support asynchronous delivery of event signals within a single thread. The event generator owns an active object, such as `CAsyncCallback`, which it informs of event signals. This active object completes itself so that its `RunL()` is executed at the next opportunity to actually send the event signal by synchronously calling the appropriate event mixin function on the event consumer. Using *Event Mixin* (see page 93) helps decouple the event generators from event mixins and provides a better encapsulated mechanism than using *Request Completion* (see page 104) directly.

References

- *Asynchronous Controller* (see page 148) extends this pattern to deal with tasks that require multiple asynchronous requests to be completed before they're done.
- *Client–Server* (see page 182) is implemented using this pattern.
- *Request Completion* (see page 104) is used to implement this pattern.
- Proactor [Harrison *et al.*, 2000] is a pattern that integrates the demultiplexing of asynchronous completion events and the dispatching of their corresponding event handlers.
- Active Object [Lavender and Schmidt, 1996] is a pattern that addresses a less constrained context to realize pre-emptive multitasking by having each active object residing in its own thread.
- [Coplien and Schmidt, 1995] contains a pattern language called Patterns of Events that deals with 'event processing in a distributed real-time control and information system'.

Asynchronous Controller

Intent Encapsulate a finite state machine within an active object to efficiently control a modest set of related asynchronous sub-tasks.

AKA None known

Problem

Context

You have a single overall task that you wish to perform that requires a modest set of asynchronous requests, or sub-tasks, to be performed in a prescribed order.

Summary

- You wish to keep your thread responsive to higher-priority event signals, such as those from end user events, whilst the overall task is being performed.
- You wish to reduce your maintenance costs by encapsulating the control of which sub-task is performed next within a single class so that future changes only need to be done in this one place.
- You wish to reduce the amount of RAM used by your software, for example by reducing the number of threads running in your process and by minimizing the code size of your solution.
- You wish to reduce your maintenance costs by reducing the time that you spend debugging your software.
- You wish to reduce the time that you need to spend testing your software looking for defects.

Description

If you've read *Active Objects* (see page 133), then your first question is going to be: 'What's the difference between a task and a sub-task?'. This is a question of granularity but the essence is that a *task* is something that other parts of your software are interested in completing whilst *sub-tasks* are the internal steps needed to achieve that end. Hence no other process is interested in whether an individual sub-task completes unless it affects the overall task. Another indication that you're dealing with a set of related sub-tasks is when they operate on a similar set of resources.

For instance, a task might be to import a series of contacts from a file into the Contacts database. This can only be achieved by performing the following sub-tasks:

1. Open the Contacts Database.
2. Read the contacts from the file.
3. Write the contacts into the Contacts Database.

Each of these sub-tasks involves making an asynchronous request to a service provider. For steps 1 and 3, the service provider is the Contacts Database whilst for step 2 it is the File Server. Using *Active Objects* (see page 133) is an obvious choice to encapsulate these requests so that you get the benefits of cooperative multitasking that they provide.

You may find it helpful here to make a distinction between *external sub-tasks*, which are asynchronous requests to an external service provider usually in another thread, and *internal sub-tasks*, which are sub-tasks performed locally that are done asynchronously to avoid impacting the responsiveness of your entire thread.

Your next question is likely to be 'Why not have an active object per sub-task?'. However, for this context that's unlikely to give the best solution because by definition the sub-tasks are closely coupled¹⁶ and there aren't a large number of them, so separating the sub-tasks into different classes is likely to reduce the maintainability of the solution in addition to increasing your code size unnecessarily.¹⁷

Another good question is 'How do I control the order in which the sub-tasks are performed?' Clearly a single class, known as the *controller*, should take responsibility for this and use some kind of Finite State Machine (FSM)¹⁸ to implement this. This is because a controller is an event handler and FSMs provide a very effective way of modeling event handling with each state starting when an asynchronous request is made and ending when the request is completed.

The most typical distinguishing feature of an FSM is that it processes a stream or sequence of input values, in which the value of each input must be interpreted in the context of previous input values. Simple examples include almost any application that responds to user input; more complex examples include parsers and communications protocols.

A number of factors determine the complexity of a state machine, including the range of possible input types; the number of states (known

¹⁶Through their mutual use of the resources and the ordering imposed between them by the overall task.

¹⁷Especially since CActive-derived classes use virtual functions, which requires a vtable.

¹⁸en.wikipedia.org/wiki/State_machine.

as sub-tasks here) that need to be performed; and the complexity of the actions which must be performed within each state.

There are a number of well-known approaches to implementing a state machine:

- *Function Tables*
Actions are dispatched via function pointers or objects that are looked up in a table based on the current state. Typically, these kinds of FSM are complex to understand, design and code. Whilst code generators can help with this, they require additional tooling support. This solution is best reserved for components which are performance-critical and for which this factor is more important than the maintenance costs. However, in this context we are generally waiting for asynchronous requests to complete and hence the time taken to make the request itself is going to be less significant than the time simply waiting for the response. Hence this approach isn't ideal for us.
- *State Pattern [Gamma et al., 1994]*
This classic approach brings all the benefits of an object-oriented design but requires a class to be used for each state. This would mean a different class for each sub-task which we've already discussed as not being ideal for this context.
- *Switch Statements and an Enumerated State*
This has the advantage of being simple to understand and implement for a modest number of sub-tasks however a C-style implementation would be poorly encapsulated.

Example

Symbian OS includes an implementation of the relatively simple HTTP protocol. This is accessed via the HTTP Transport Framework, notably `RHTTPSession` and `RHTTPTransaction`, however, these simply provide a convenient interface to the actual protocol itself which is hosted in the Socket Server as part of the Communication Infrastructure.¹⁹

The protocol is functionally decomposed into a number of tasks that mirror those required by clients of HTTP. These tasks include:

- establishing an HTTP connection with a remote device
- listening for incoming connections from remote devices

¹⁹See [Campbell et al., 2007] for more details of this.

- reading in data from remote devices and passing it to a client
- taking data from a client and passing it to a remote device.

All of these tasks can be broken down into sub-tasks as shown in Figure 5.5. For the purposes of this example, we'll just focus on the connection task which can be represented by the following state diagram which reflects the need to first find out the IP address of the remote device by doing a DNS lookup followed by the actual connection attempt:

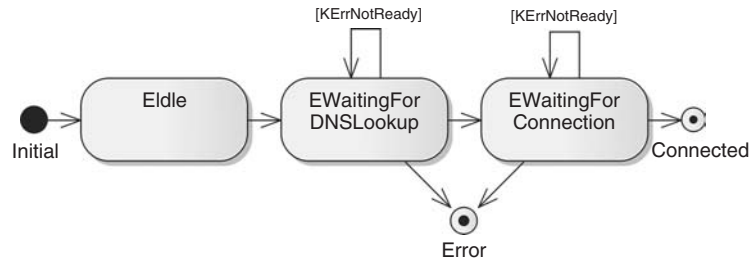


Figure 5.5 Example of the *Asynchronous Controller* pattern (connection state diagram)

Solution

The fundamental design is to encapsulate both the event handling and the FSM into a single *asynchronous controller*, implemented as an active object. *Active Objects* (see page 133) provides a native model for handling asynchronicity without the complexity of explicit multi-threading whilst providing a convenient class to contain a moderately complex FSM implemented using enumerated constants and a switch statement. This offers an effective compromise between the over-simple and the over-complicated whilst avoiding the drawback of a non-object-oriented C-style approach.

Structure

Whether the FSM is externally or internally driven, each state represents a single sub-task which starts with an asynchronous request and ends when it is completed. This is represented by Figure 5.6.

The asynchronous controller will have at most three places where the state it is in makes a difference to its behavior and hence a switch statement is needed:

- Within its `RunL()` where it uses the state it is in to determine which event signal it has received and so what action it should take next. Usually this will be a transition to the next state via a private

function that issues the asynchronous request corresponding to the next sub-task.

- If your task needs specific error handling then you should add a `RunError()` which collects all the error-handling code. A `switch` may then be useful to determine which sub-task failed.
- You will normally need a `switch` statement within your `DoCancel()` to ensure the correct cancellation request is sent for the current sub-task.

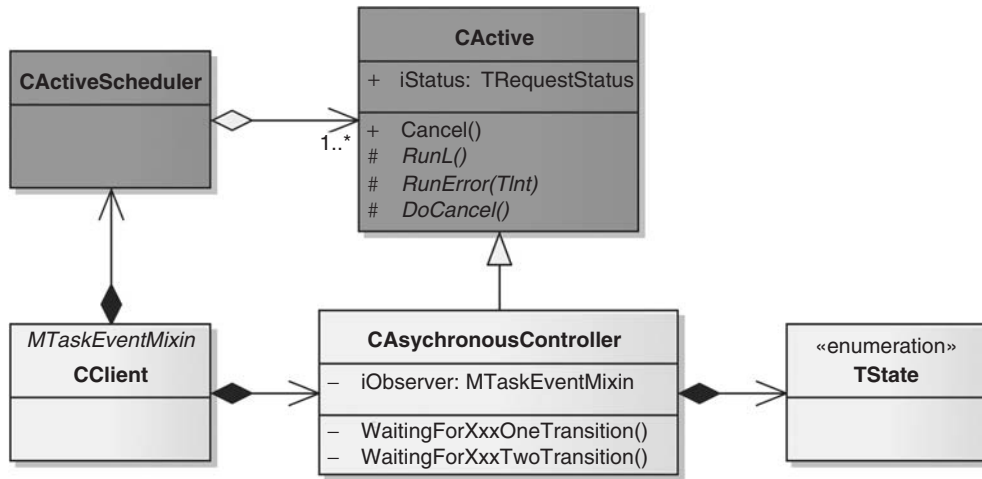


Figure 5.6 Structure of the *Asynchronous Controller* pattern

Dynamics

The entire task that you wish to achieve is represented by an active object hence creating the object usually starts the task and destroying it cancels the task if it is still running. *Event Mixin* (see page 93) is used to provide a way for the object that starts the overall task to be told when it has completed (see Figure 5.7).

The `RunL()`, `RunError()` and `DoCancel()` methods of the asynchronous controller give rise to the dynamic behavior of the FSM and enact the state transitions and actions. Typically the `switch` statement acts on the enumerated state variable and dispatches a single action via a method invocation which should be synchronous and short running.²⁰ If there are further sub-tasks to perform then the `switch` will also cause a transition to the next state and issue the associated asynchronous request before waiting to handle the next incoming event.

²⁰Since active objects are cooperatively multitasked.

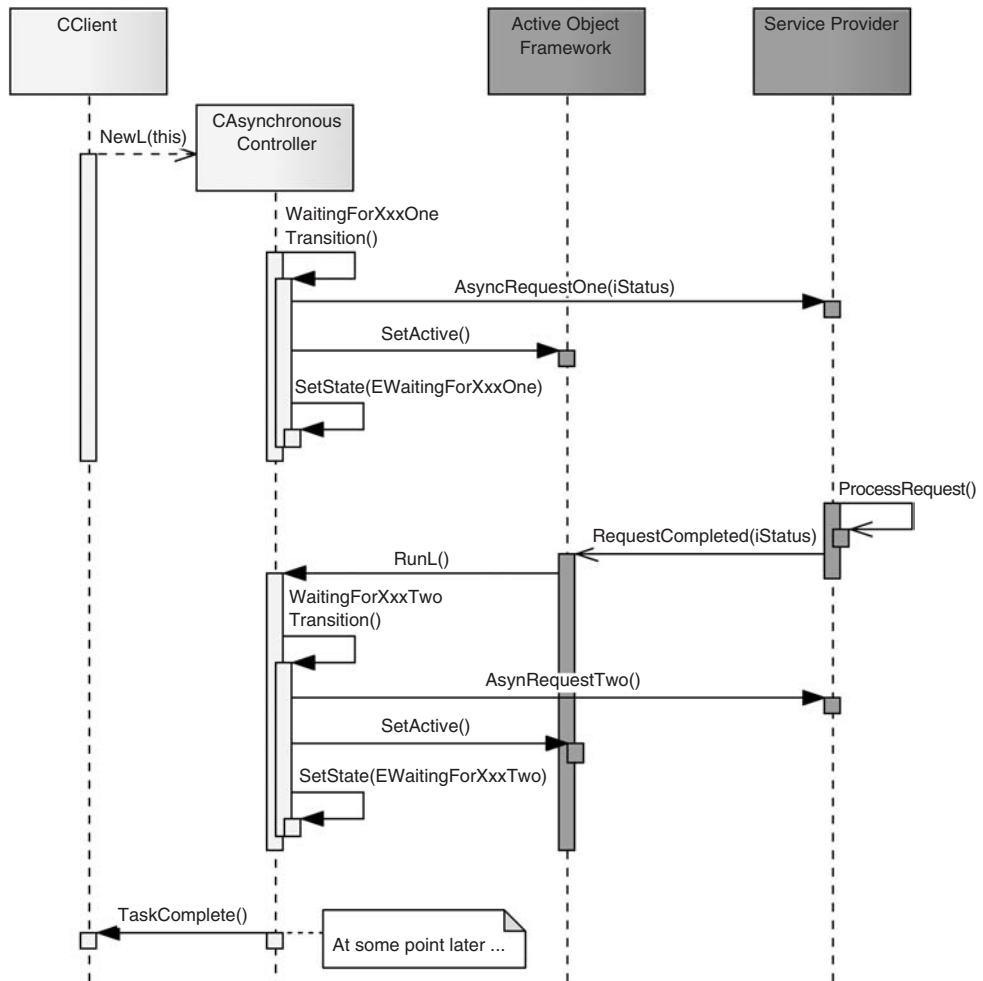


Figure 5.7 Dynamics of the *Asynchronous Controller* pattern

Implementation

Your asynchronous controller will look something like this code:

```

class CMyAsyncController : public CActive
{
public:
    static CMyAsyncController* NewL(MTaskEventMixin& aObserver);
    ~CMyAsyncController();
private:
    CMyAsyncController();
    void ConstructL();

    // from CActive

```

```

void RunL();
TInt RunError(TInt aError);
void DoCancel();

// State transistions
void WaitingForXxxFirstTransition();
...
void WaitingForXxxLastTransition();
private:
// Used to signal the completion of the overal task
MTaskEventMixin& iObserver;

// For simplicity we show just a single service provider that is used
// by each of the sub-tasks
RServiceProvider iProvider;

// Defined with CMyAsyncController so that the individual enums aren't
// in the global namespace
enum TState
{
    EInitial = 0, // Must be 0 so that it is initialized for us by CBase
    EWaitingForXxxFirst,
    ...
    EWaitingForXxxLast,
};

TState iState;
};

```

This class is then implemented as follows. Just the `ConstructL()` function is shown here as the `NewL()` and the constructor²¹ are standard for an active object using two-phase construction.

```

void CAsyncController::ConstructL()
{
    // As an example we assume that all the sub-tasks operate on
    // the same service and so a handle to it is opened here.
    User::LeaveIfError(iProvider.Open());

    // Start the FSM
    WaitingForXxxFirstTransition();
}

```

The state transition functions all follow a similar style so only one is shown here. When creating your specific implementation the contents of these functions will largely be determined by the exact FSM that you require.

```

void CAsyncController::WaitingForXxxFirstTransition()
{
    ASSERT(!IsActive());
}

```

²¹Note that you don't need to set the `iState` to be `EInitial` in the constructor since `CBase` classes are zero-initialized when they are created which gives us the correct value.

```
// Need to set this first in case the async request Leaves
iState = EWaitingForXxxFirst;

iProvider.AsyncRequest(iStatus);
SetActive();
}
```

When the above request completes, the `RunL()` is called and you need to determine exactly which asynchronous request has been completed by switching on `iState`. Interestingly, the function is an example where *Fail Fast* (see page 17) is used (to handle either an illegal or unrecognized state in the switch since that indicates a programming fault has occurred) in addition to *Escalate Errors* (see page 32) (to handle errors from the state transitions). This works because they are each being used for different errors.

```
void CAsyncController::RunL()
{
    // All errors are handled in RunError() so ...
    User::LeaveIfError(iStatus);

    switch(iState)
    {
    case EWaitingForXxxFirst:
    {
        // Move on to the next state
        WaitingForXxxNextTransition();
        break;
    }

    ... // Other states are handled here

    case EWaitingForXxxLast:
    {
        iObserver->TaskComplete(KErrNone);
        break;
    }
    case EInitial:
    default:
        // Illegal or unrecognized state
        Panic(EUnexpectedAsyncControllerRunLState);
        break;
    } // End of switch statement
}
```

The `RunError()` method is used to handle any errors escalated from the `RunL()`:

```
TInt CMyActive::RunError(TInt aError)
{
    TBool errorResolved = EFalse;
    switch(iState)
```

```

    {
    case EWaitingForXxxFirst:
    {
        // For example we can resolve KErrNotFound errors here ...
        if (aError == KErrNotFound)
        {
            // Resolve error such as by retrying
            WaitingForXxxFirstTransition();
            errorResolved = ETrue;
        }
        break;
    }

    ... // Other states are handled here

    case EInitial:
    default:
        // Illegal or unrecognized state
        Panic(EUnexpectedAsyncControllerRun1State);
        break;
    }

    if (errorResolved)
    {
        return KErrNone;
    }
    // Else the error hasn't been resolved so end the task
    iObserver->TaskComplete(aError);

    return aError; // Pass the error on up
    }

```

The `DoCancel()` method should be used to cancel the asynchronous request for the current state. Here we don't bother sending `KErrCancel` to the observer since we follow the semantics of `CActive::Cancel()` which is that this is called when no one cares about the task any more.

```

void CAsyncController::DoCancel()
{
    switch(iState)
    {
    case EWaitingForXxxFirst:
    {
        iProvider.CancelAsyncRequestOne();
        break;
    }

    ... // Other states are handled here

    case EInitial:
    default:
        // Do nothing
        break;
    }
}

```

Finally the destructor just calls `Cancel()` and any resources used are cleaned up:

```
CMyActive::~CMyActive()
{
    Cancel();

    // Other cleanup code
    iProvider.Close();
    ...
}
```

Consequences

Positives

- For a moderate number of sub-tasks (less than ten might be a reasonable rule of thumb) this pattern retains the intuitive simplicity of a C-style, `switch`-statement implementation with the added benefits of encapsulation.
- Your thread remains responsive whilst the overall task is being done. For applications, this will mean your UI continues to service end user requests.
- Using *Active Objects* (see page 133) as part of this pattern gives us all the advantages of that pattern when compared to a multithreaded alternative including the RAM savings, reduced maintenance and testing costs and the flexibility to prioritize other tasks over the one managed by the asynchronous controller.
- By wrapping multiple asynchronous requests into a single active object, we mitigate one of the disadvantages of *Active Objects* (see page 133) – that the active scheduler doesn't scale well to cope with many active objects – by reducing the number of active objects needed.

Negatives

- This pattern doesn't scale well when the number of sub-tasks increases.
- Using *Active Objects* (see page 133) as part of this pattern gives us all the disadvantages of that pattern when compared to a multithreaded alternative caused by its cooperative scheduling approach.

Example Resolved

Here we just show the how the connection task is resolved. An asynchronous controller is created to manage the task with a separate state for both of the asynchronous requests it need to make to the DNS server

to look up the IP address of the remote device and to establish the TCP connection to that IP address.

The way that the asynchronous controller is started and stopped is slightly different from the pattern described above. For clarity, the task is not begun in the constructor but instead a separate function has been added that begins the connection process – `ConnectL()`. At the end of the task, an additional state has been added to deal with the non-trivial work that needs to be done once the connection has been made. No asynchronous request is made at this point but a transition method has been added to keep the `RunL()` consistent.

```
class CSocketConnector : public CActive
{
public:

    static CSocketConnector* NewL();
    virtual ~CSocketConnector();

    void ConnectL(MSocketConnectObserver& aObserver,
                  const TDesC8& aRemoteHost,
                  TUint aRemotePort);
private:
    CSocketConnector();

    // methods from CActive
    virtual void RunL();
    virtual void DoCancel();
    virtual TInt RunError(TInt aError);

    // State transition methods
    void WaitingForDnsLookupTransitionL();
    void WaitingForConnectionTransitionL();
    void ConnectedTransitionL();
private:
    enum TConnectState
    {
        EIdle = 0,

        // A connection has been requested so a DNS lookup needs to be
        // initiated to find the IP address of the remote host.
        EWaitingForDnsLookup,

        // The IP address of the remote host has been found so we need to
        // initiate a TCP connection to that remote host.
        EWaitingForConnection,

        // Task complete
        EConnected,
    };
private:
    TConnectState iState;
    MSocketConnectObserver& iObserver;
```

```
// Service Provider for the EWaitingForDNSLookup state
RHostResolver iHostResolver;

// Service Provider for the EWaitingForConnection state
CSocket* iSocket;
...
};
```

The following function starts the whole task going. You should note that it takes as parameters all the information needed to complete the whole task rather than just the first sub-task. This is a common phenomenon since the client of the asynchronous controller only knows about the overall task and has just this one opportunity to provide all the necessary information.

```
void CSocketConnector::ConnectL(MSocketConnectObserver& aObserver,
                                const TDesC8& aRemoteost,
                                TInt aRemotePort)
{
    // Use Fail Fast (see page 17) to catch faults in the client
    __ASSERT_DEBUG( iState == EIdle, Panic(EBadSocketConnectorState) );

    // Store parameters for later
    iObserver = aObserver;
    iHost = HBufC::NewL(aRemoteHost.Length());
    iHost->Des().Copy(aRemoteHost);
    iPort = aRemotePort;

    // Start the task
    WaitingForDnsLookupTransitionL();
}
```

The RunL() implements the normal flow through the FSM which is fairly simple for this case simply moving onto the next state in sequence:

```
void CSocketConnector::RunL()
{
    // Leave if there has been an error
    User::LeaveIfError(iStatus.Int());

    switch(iState)
    {
    case EWaitingForDnsLookup:
    {
        WaitingForConnectionTransitionL();
    } break;
    case EWaitingForConnection:
    {
        ConnectedTransitionL();
    } break;
    case EIdle:
    default:
```

```

        Panic(EBadSocketConnectorState);
        break;
    } // End of switch statement
}

```

The error paths through the FSM are slightly more interesting in that in the `EWaitingForDnsLookup` and `EWaitingForConnection` states it's worth re-trying if `KErrNotReady` is received. Otherwise an error ends the task early so the observer needs to be informed.

```

TInt CSocketConnector::RunError(TInt aError)
{
    TBool errorResolved = EFalse;

    switch(iState)
    {
    case EWaitingForDnsLookup:
    {
        if (aError == KErrNotReady)
        {
            WaitingForDnsLookupTransitionL();
            errorResolved = ETrue;
        }
        break;
    }
    case EWaitingForConnection:
    {
        if (aError == KErrNotReady)
        {
            WaitingForConnectionTransitionL();
            errorResolved = ETrue;
        }
        break;
    }
    case EConnected:
    {
        // This is a valid state to have an error in but there's nothing we
        // can do about it
        break;
    }
    case EIdle:
    default:
        Panic(EBadSocketConnectorState);
        break;
    }

    if(errorResolved)
    {
        return KErrNone;
    }
    iObserver->HandleConnectError(aError);
    return error;
}

```

`DoCancel()` simply ensures we send a cancellation to the correct service provider:


```

void CSocketConnector::DoCancel()
{
    switch(iState)
    {
    case EWaitingForDnsLookup:
    {
        // DNS lookup is pending so
        iHostResolver.Cancel();
    } break;
    case EWaitingForConnection:
    {
        if(iSocket)
        {
            // Connection is pending - cancel and delete the socket
            iSocket->CancelConnect();
            delete iSocket;
            iSocket = NULL;
        }
    } break;
    case EConnected:
    case EIdle:
    default:
        // Do nothing...
        break;
    }
}

```

All the other functions not mentioned are either as per the standard pattern given above or, as in the case of the transition functions, don't help to illustrate this pattern.

Other Known Uses

- *Mass Storage*
This pattern is used within the File System to handle the ongoing task of managing the bulk-only, mass-storage transport and communications with the SCSI protocol. Its asynchronous controller has states such as:
 - waiting for a Command Block Wrapper (CBW) packet
 - sending a Command Status Wrapper (CSW) packet
 - reading and writing data.
- *SyncML*
The SyncML Sync Engine uses the 'Long-Running Active Object' variation of this pattern when processing messages. Although messages could be parsed and processed synchronously, their processing is potentially long running. Therefore, the large parsing task is broken

down into smaller more manageable parsing sub-tasks and driven by an asynchronous controller.

Variants and Extensions

- *Self-Completing Active Object*

Some active objects need to complete their own requests to induce an asynchronous callback. The main reason for doing this is that it yields control back to the active scheduler so that other, perhaps higher-priority, active objects can run. This is done via the following snippet of code:

```
void CMyAsyncController::SelfComplete(TInt aError)
{
    SetActive();
    TRequestStatus* s = &iStatus;
    User::RequestComplete(s,aError);
}
```

This is not a variation of *Active Objects* (see page 133) since if you only need to self-complete once and don't have any other use for the active object then a better alternative is to use `CAsyncCallback` provided by the operating system. If you call self-complete more than once, you've probably got some kind of FSM going on and hence should use this pattern.

- *Long-Running Active Object*

While handling external event signals is a common use case for this pattern your asynchronous controller can just as well be driven by internal sub-tasks. Most structured actions can be recast in terms of a succession of sub-tasks; each portion of the whole action is treated as a sub-task, and the asynchronous controller tracks the progress through them. The main reason you'd want to do this is because the overall task would take too long to do purely within the `RunL()` of an active object since it would prevent all the other active objects from running. Hence the task is broken up into chunks and regular yield points inserted as per the self-completing active object variation above. For example, the re-calculation of values in a spreadsheet could mean thousands of evaluations and hence might be better restructured as 10 sub-tasks each doing hundreds of evaluations. Note that you need to choose the step size with care because the cost of this solution is increased execution overhead which is proportional to the number of sub-tasks. This pattern is clearly suited to addressing this problem as well given a suitable FSM that divides up the original task. In addition Symbian OS provides a specialized class to help with specifically this issue – `CIdle`.

- *Other FSM Designs*

This pattern uses a fairly simple FSM design with `enums` and `switches` however it can also be generalized to work with other state patterns such as those described in the Models for Object-Oriented Design of State (MOODS) section of [Vlissides *et al.*, 1996] and the State Patterns of [Martin *et al.*, 1998].

References

- *Active Objects* (see page 133) provides the foundation for this pattern.
- The State Patterns of [Martin *et al.*, 1998] and the MOODS of [Vlissides *et al.*, 1996] provide alternatives to the `enum-switch` approach to FSMs taken here.

6

Providing Services

We define a *service* to be a collection of useful actions that are packaged up by a vendor, the *service provider*, and supplied for use by software components, the *clients*. An *action* can be anything that is of benefit to a client. The dialog or conversation between the service and a client is known as a *session* which is started at a certain point and then stopped. An established session often involves more than one message in each direction. When a client asks the service for an action to be performed on its behalf this is known as a *service request*. When the service has finished performing the action it sends a *request complete* message in response (this might be implemented as a simple function return).

A session is often *stateful*: either the client or, more commonly, the service has to maintain information about the session history in order to be able to communicate. In *stateless* sessions, the communication consists of independent requests with responses. An example of a stateful software service is a component that provides read and write access to a database since the service may have to track any changes to the database up until the client calls commit or rollback. An example of a stateless service is a math library which provides actions such as returning a random number or the square root of a number.

A key reason for having services is that they offer software reuse which can be used to build software more quickly, more efficiently and more reliably. The alternative to using a service is to create your own stand-alone copy for each client which quickly becomes costly to develop. The long term costs of duplicated code can also be significant as not only is there more code to support but any additional functionality must also be replicated. As if that wasn't enough, the replication increases code size and the associated RAM impacts could easily be significant on a resource-constrained device such as a smartphone.

A well-designed service has the following properties:

- *Cohesion* – a measure of how strongly-related and focused the various actions offered by the service are. A highly cohesive service enforces its logical boundaries.
- *Separation of concerns* – the service overlaps in functionality as little as possible with other services available to clients.
- *Encapsulation* – the design decisions on which the service is based are the ones that are most likely to change are hidden from clients. Well-encapsulated services provide a stable interface which shields clients from changes in how the service is implemented.
- *Extensibility* – how well the service supports future growth whilst remaining backward compatible. Backwards compatibility is where the clients of an old version of the service can run on later versions of the service without being changed.
- *Usability* – an indication of how easy the service is to use which generally reflects the elegance and clarity with which the interface to the service has been designed.

These properties focus on making life easier for the client developers; the more clients a service has, the more important these properties become.

There are two main ways in which the actions of a service can be provided to clients: *synchronously* and *asynchronously*. An action offered by a service may take any amount of time from nanoseconds for a couple of instructions to many seconds or more. If the time taken is not significant, a client may wish to choose to make a synchronous request to the service that blocks the client's execution until the action has been completed. A function which simply calculates and then returns a value is an example of this. Using a synchronous request allows a client to simplify its design. However where the action may take a considerable amount of time an asynchronous request may be needed to guarantee some quality of service, such as responding to events generated by the end user. Figure 6.1 illustrates the difference between the two types of actions.

The main difficulty with asynchronous requests is that they introduce additional execution paths for clients to deal with. This is particularly difficult for them as they are not able to predict in advance which execution path will be taken at run time. To help reduce complexity for clients, we recommend you follow a consistent approach for asynchronous requests as follows.

The asynchronous request always has two responses: the immediate synchronous 'request made' response that simply indicates a request has been received by the service; and the asynchronous 'request complete'

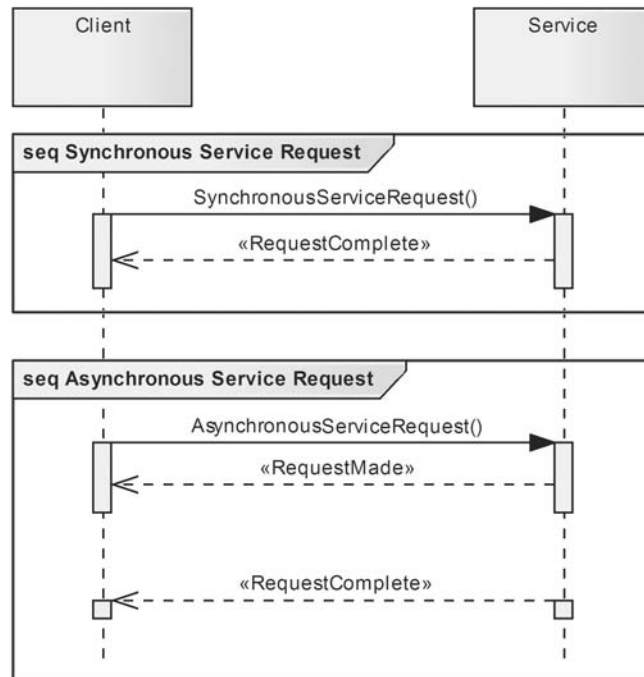


Figure 6.1 Synchronous and asynchronous service requests

response at some point later that indicates that the request has actually finished. All errors in carrying out the service request should be returned through the ‘request complete’ response. This helps prevent the duplication of error-handling code for clients. This may mean the request is in fact completed immediately if an error occurs whilst making the initial service request.

We also need to consider how to allow clients to cancel an asynchronous request. This is where a client makes a synchronous request to ask that a previous asynchronous request should be completed as soon as possible. Figure 6.2 shows the most obvious sequence diagram for a cancellation in which the original asynchronous service request is prevented from succeeding and completes with a `KErrCancel` to show that it was stopped prematurely.

An alternative sequence (see Figure 6.3) is that the client sent the cancel service request too late and the service has finished performing the action originally requested. The result is that the asynchronous service request completes with `KErrNone` to indicate the action originally requested was performed.

There are even more alternative execution sequences, such as the asynchronous service request completing with an error that isn’t `KErrCancel` or `KErrNone` since the action had time to complete

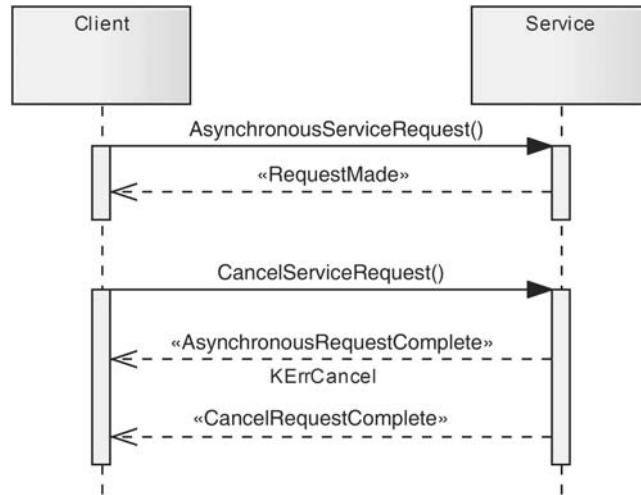


Figure 6.2 Asynchronous service request stopped early enough

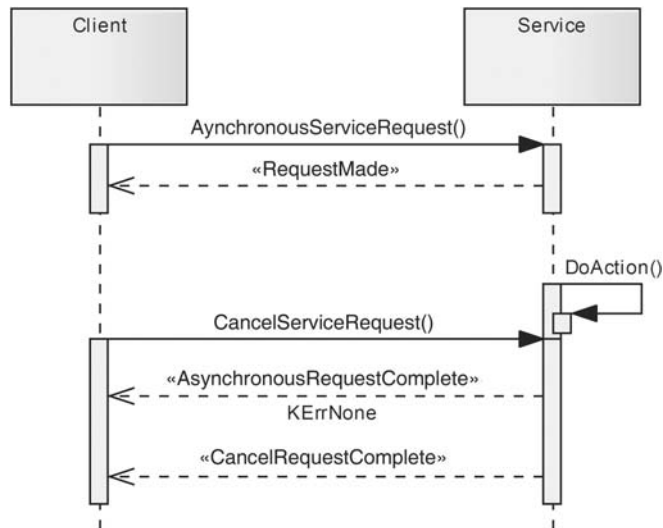


Figure 6.3 Asynchronous service request couldn't be stopped

but some other error intervened, or the client has sent a cancel when there isn't a service request outstanding.

We recommend service providers follow a consistent approach for canceling asynchronous requests as follows:

- Cancellation requests are full requests in their own right separate from the asynchronous request they are canceling. Hence, there is a

‘request complete’ response for the cancellation separate from one for the original asynchronous request.

- Cancellation requests are always synchronous. This is to ensure that a cancellation can be used in the destructor of an object to clean up the resources it is using. It doesn’t make sense to try to stop an asynchronous request with another asynchronous request.
- The asynchronous request being canceled must be completed before the cancel request itself is completed otherwise the client may have been deleted prior to the completion of the second request completion.
- The error message returned with the completion of the asynchronous request should reflect whether or not the original request was successful. That is, if the request went ahead and the action was performed successfully then `KErrNone` should be used; if the action was stopped prematurely, `KErrCancel` should be used.

It is important to remember that cancel does not mean that any action that the asynchronous request has already performed should be undone. If you need such rollback semantics then canceling is only the first step you need to take.

Note that as a client of a service you may not need to deal with each and every message from the service yourself and hence it might appear as if this approach is not being used. This often results from services that provide helper objects, based on the Proxy pattern [Gamma *et al.*, 1994], that clients use to access the service indirectly. One example of this is `CActive::Cancel()`, described in *Active Objects* (see page 133) where calling this function does not result in the object’s `RunL()` being called to handle the completion of the original request. This makes it appear as if the original request wasn’t completed. The reason for this is that the Symbian OS active object framework chooses to simplify the majority of active objects by assuming they wish to ignore the completion of an asynchronous request after it has been canceled.

The patterns in this chapter describe some techniques for providing services following these principles. The simplest is *Client-Thread Service* (see page 171) which describes how to provide a service that executes within the context of the client’s thread. This allows for a simple design but is not suitable for more complex services which need to provide access to a shared resource or must check the security credentials of clients before accepting requests from them.

Services which require such functionality should consider using *Client-Server* (see page 182) to solve their problem. This pattern provides support for clients in separate threads by allowing them simple and synchronized access to the resources and actions they require. This pattern

is also well suited to enforce any security policies required by a service provider.

Last but not least, *Coordinator* (see page 211) describes a technique used by services to allow multiple interdependent clients to receive notification of state changes that they need to react to in a structured manner.

Client-Thread Service

Intent Enable the re-use of code by rapidly developing a service that executes independently within the thread of each client.

AKA Shared Library¹

Problem

Context

You wish to develop a service that can be re-used by a number of components, with clients of the service able to operate independently of each other.

Summary

- It is expensive to create stand-alone code for each application that needs it.
- Maintenance of multiple copies of such code usually doesn't scale well and is defect-prone.
- Duplication of common code is memory inefficient in terms of both code size and RAM usage.

Description

Novice software developers often re-write similar code for each application – code that essentially performs the same actions and which may even be copied and pasted between projects.

This is a problem, and not just because re-writing code is expensive (and boring). Simplistic solutions such as copying and pasting do not scale well, because each new copy increases the cost of maintenance. In addition, the resources consumed across the system due to the additional code size, resulting from all the duplicated code, is unacceptable on constrained Symbian OS devices.

This problem occurs in all software problem domains, ranging from device creation to third-party application development.

Example

Calendar and contacts databases, and indeed any database, usually store their data using the most readily available database service – it's much

¹ en.wikipedia.org/wiki/Shared_Library.

faster to use DBMS (or Symbian SQL from Symbian OS v9.3 onwards) than to invent or port a database engine! In order to exchange contact and calendar data with personal information management (PIM) applications on other systems, a common data exchange format and a parser for converting the data between internal and external formats must exist.

A naïve developer might re-write this parser as part of each application that needs it. This would include at least the PIM UI applications (e.g. Calendar and Contacts) and data synchronization components, such as SyncML. Any third-party software that needs to export PIM entries would also need to have its own parsers.

Developing a private version which just does the few things that are initially required might, at first, appear easier than creating a generic service or understanding a service written by someone else. Unfortunately, each implementation would contribute additional unnecessary code to the device and hence use more RAM.² If there was a defect discovered then this may or may not also apply to the other instances – certainly the problem might need to be checked in all of them. If the underlying data-exchange standard was changed, then again the code would need to be updated in each place it was used.

Solution

The solution is to deliver a single copy of what would otherwise be duplicated code as a re-usable client-thread service (CTS). The principal variant of this pattern is a service hosted in a DLL that is loaded into a client at run time.

This pattern is the default, or preferred, choice for implementing a service (where it is applicable) because it is the simplest, in terms of implementation, and the most lightweight, in terms of memory consumption.

Structure

Services based on this pattern run entirely within the client thread. For example, Figure 6.4 shows a CTS packaged in a DLL to which the client statically links at compile time.³ To access the CTS, the `client.exe` calls a function exported from the `cts.dll`. Remember that you should design services to be extensible, so that they can be updated and extended without affecting clients. In this case, since the CTS is being used via its exported functions, it is these functions and the associated objects that you need to make future-proof.

Each process that uses the CTS loads its own copy of the executable code contained in `cts.dll` into its virtual address space.⁴ However, all

²At least on a non-XIP device, currently the most common type of Symbian OS device.

³Although the code is actually loaded at run time.

⁴At the time of writing, the most commonly used memory model is the Multiple Memory Model which only loads executable code into RAM once no matter how many processes load it. See [Sales, 2005] for more information.

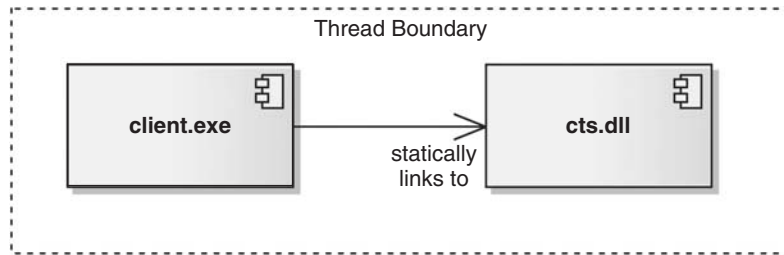


Figure 6.4 Structure of the *Client-Thread Service* pattern

of the objects and writable data created using the `cts.dll` code are located on the executing thread's local heap.⁵ Hence, each client thread has its own entirely independent instantiation of the CTS.

This structure gives only limited opportunities to enforce security checks between the `client.exe` and the `cts.dll`. If this is a requirement for you, see Chapter 7, in particular *Buckle* (see page 252). However, since this pattern allows clients to operate independently of each other, you at least do not need to worry about one client of the CTS attacking another client.

Dynamics

By the time the `client.exe` first uses it, the CTS code must have been loaded into the client's process. Usually this does not take a significant proportion of time but for large executables (such as those that are hundreds of KB)⁶ it might start becoming a factor.⁷

Once the code is loaded the run-time behavior can be as straightforward as a function call from the client to an object, possibly followed by the object returning some value; both cases are shown in Figure 6.5. Of course, more complex scenarios are possible – the service may be supplied by a number of objects, which may make calls to each other and to other code.

Implementation

A CTS is provided through one or more DLLs to which the client statically links at compile time. To implement this, service providers must export the header file, containing the public APIs for the CTS, to the system include directory (`\epoc32\include`) from their `bld.inf` file. Clients access the actions of a service by specifying the service header files in

⁵ Assuming you do not use writeable static data (WSD) in DLLs since it costs 4 KB of RAM per process that uses the CTS. See *Singleton* (page 346) for further discussion.

⁶ Measured using the `urel` version of your component compiled for your target hardware.

⁷ Where demand paging is supported, the load time for large executables is significantly less of an issue.

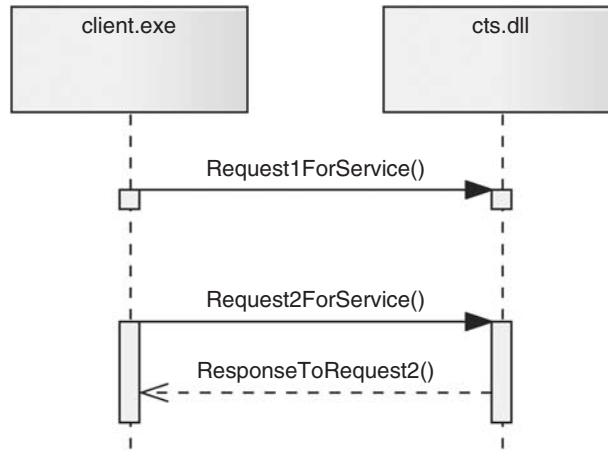


Figure 6.5 Dynamics of the *Client-Thread Service* pattern

a `#include` statement, using the classes they need, and specifying the linked `cts.dll` in their `MMP` file `LIBRARY` statements.

Appropriate platform security capabilities must be specified in the `MMP` file of the `cts.dll`. The service provider should declare all of the capabilities used by its clients or, if the clients are not known at build time, as many capabilities as the signing authority will grant you.⁸ Note that this must also include the capabilities required to access the APIs actually called from the `cts.dll`. In practice, most developers are not able to get all capabilities (e.g. manufacturer capabilities) so there may be a limit on which clients can use services provided through this pattern.

One of the most important parts of specifying a service is the definition of the service API. The API typically comprises one or more classes. The exported public interface should be clear and expose as little implementation detail as possible. When creating the API, you should give some consideration to how you will provide binary compatibility in the event of API extension.

The Symbian Developer Library includes documentation and examples for creating DLLs that you should find useful.

Consequences

Positives

- It is simpler for the service provider to implement than almost any other alternative. Only one instance of the code needs to be

⁸Under the Symbian OS platform security model, a DLL is only trusted to load into a client process if it has at least the capabilities of that process (the DLL loading rule). Therefore in order to be trusted to be loaded into *any* client (which may have any capabilities), a service DLL must have *all* capabilities (see [Heath, 2006, Section 2.4.5]) although usually we settle for All -TCB.

developed, reducing the development time and cost for all its clients. As services are provided in one package, to a large extent the CTS can be maintained, improved and upgraded independently of its clients.

- Services based on this pattern are relatively easy to extend, provided that thought is put into this when you define the DLL interface. Of course, care must still be taken to ensure that upgraded services maintain compatibility with previous versions.
- It is easy for the client to grow over time: as a CTS often provides more functionality than is initially used by a single client, it's not uncommon for it to remain fit for purpose as the client expands its role.
- Since the service is tested against more users and over a longer time period than stand-alone instances, it is likely to be of high quality.
- The variants of this pattern require little 'infrastructure' code to implement, thus minimizing the code size.
- There is only a single copy of code in physical memory and, typically, there is less code generated in the first place, so less RAM is used.
- It is quick to start up. The start-up time consists almost entirely of the cost of loading the service code into memory, which normally doesn't take a significant proportion of the time since most DLLs are less than 100 KB on a mobile device. Unlike for other mechanisms, there is no process context switch and little cost in time for the service infrastructure to initialize itself.
- The system remains responsive: all functions are accessed from within the same thread, so there are no direct costs in switching thread or process contexts (there may be indirect costs, for example if the service uses other services that are based on different service patterns).

Negatives

- This pattern cannot mediate access to shared resources. It does not include mechanisms to provide serial access to a resource, or to deal with concurrency issues such as the atomicity of an operation, because it assumes these issues will never occur. It is possible to graft such support onto this pattern, but the cost of doing so often makes the use of other patterns a better solution.
- The CTS cannot securely verify a client's identity or its capabilities. If you need to be able to do this then you should consider an alternative pattern, such as *Client-Server* (see page 182).

- Symbian OS is designed to run in a resource-constrained environment. For memory efficiency reasons, the operating system imposes some further constraints on the use of this pattern:
 - Symbian OS uses link-by-ordinal rather than link-by-name, which requires additional effort to be taken to maintain binary compatibility.
 - Symbian OS supports writeable static data (WSD) in DLLs from Symbian OS v9, but with some cost caveats [Willee, Nov 2007]. This is discussed in more detail in *Singleton* (see page 346) but the cost can be as high as 4 KB per process that uses the CTS. If you are porting a CTS service from operating systems that does make heavy use of WSD then it may take more development effort to re-design your CTS to avoid the use of WSD than it would to use another pattern entirely to provide the service. [Willee, Nov 2007, Section 2.3] discusses this in more detail.

Example Resolved

Symbian OS provides a parser for Calendar and Contacts data shared between several applications. The *Versit* service allows clients to import and export data formatted according to the vCard and vCalendar standards.⁹ This parser is provided through a client-thread service.

This pattern is the obvious choice for the parser design because it's lightweight, simple and well understood. The obvious reasons why the design might not be suitable do not apply:

- The parser does not need to mediate access to any shared resources. Instead it relies on other services to do this. For example, if the data to be parsed is in a file, access is mediated at a higher layer by the Contacts engine, the Calendar engine or the File Server.
- The parser does not need to authenticate its clients directly. Any authentication that needs to be done is performed by the service that mediates access to the secure resource such as the File Server or Contacts engine.

The parser is used by (at least) the Contacts and Calendar models, the Bluetooth phonebook access profile, and the J2ME implementation. Instead of having the functionality in many places, each requiring its own maintenance and occupying its own memory, the parser code has been developed once and is maintained in one place. When the parser is upgraded, all clients get the benefit. The total uncompressed size of the parser binaries is around 33 KB, so the ROM savings across just the Symbian OS components which use the parser is around 100 KB.

⁹These standards are prescribed by the Versit consortium (www.imc.org/pdi).

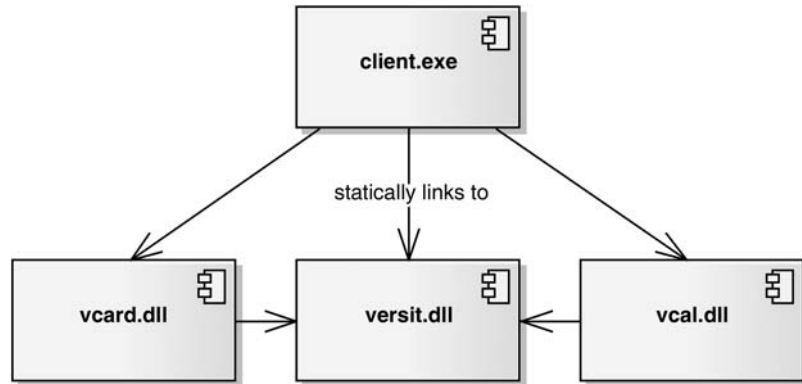


Figure 6.6 Structure of the Versit Client-Thread Service

The services provided by Versit are packaged in three DLLs that are loaded into the client process (see Figure 6.6). `versit.dll` exports the service's generic base class `CVersitParser`. This class is extended to provide specific services for parsing of vCards (exported from `vcards.dll`) and vCals (exported from `vcal.dll`) and their sub-entities, such as `vEvents`, `vTodos`, etc. `CVersitParser` has also been designed so that it can be extended to parse other data formats with similar requirements.

Client Implementation

To use the vCard service, a client includes the service provider's header in a CPP file:

```
#include <vcards.h>
```

Clients can then instantiate the vCard parser service using its `NewL()` method, internalize the vCard from a stream, and then start doing operations on it:

```
CParserVCard* vCardParser = CParserVCard::NewL();
// Create a handle to the input stream, e.g. from a file ...
vCardParser.InternalizeL(stream) // To import a vCard
```

In the client project's MMP file, the location of the header file is specified by a `SYSTEMINCLUDE` statement and the relevant link libraries are listed in one or more `LIBRARY` statements:

```
SYSTEMINCLUDE \epoc32\include
LIBRARY vcard.lib versit.lib
```

Service Provider Implementation

The API is typically defined in a header file that is exported to a public area via the associated project `bld.inf` file. The API comprises one or more classes that are declared as exported¹⁰ in a header file. Note that a class is exported if it has at least one method whose declaration is imported using `IMPORT_C` and whose definition is exported using `EXPORT_C`.

The relevant part of the MMP file for the vCard parser DLL is shown below. This is fairly typical – most service providers based on this variant will be very similar.

```
// VCARD.MMP
target          vcard.dll
targettype      DLL
CAPABILITY      All -TCB
UID             0x1000008D 0x101F4FC6
SOURCEPATH      ../src
userinclude     ../inc
systeminclude   /epoc32/include
source          VCARD.CPP
library         versit.lib euser.lib estor.lib baf1.lib
```

The important statement here is `TARGETTYPE DLL`, which tells the build tool chain that it should create a DLL (rather than an EXE or some other target) and export the required CTS binary interface. The `LIBRARY` statement specifies the statically linked dependencies of this DLL. For example, `vcard.dll` links to `versit.dll`, in order that `CParserVCard` can derive from `CVersitParser`.

The `CAPABILITY` statement specifies the capabilities that this DLL has been granted. A DLL must have at least the capabilities of all of its client processes or they will not be able to load it.¹¹ Device creators usually give service DLLs the capabilities `ALL -TCB` (as above) in order to ensure the service can be loaded by any process. A third-party developer may not be able to get the necessary capabilities – in which case they might use an alternative variant (such as supplying the CTS as a static LIB) or allow clients to compile their own versions of the DLL with just the capabilities they need.

The other statements are important too, but as they are less relevant to the discussion and are well documented in the Symbian Developer Library, they are not covered further in this book. More information, including a detailed architectural overview of Versit, is also given in the Symbian Developer Library.

¹⁰Note the word ‘export’ has two meanings here: one refers to header files being exported for use during development and the other to classes and functions that are exported from an executable for use at run time.

¹¹See [Shackman, 2006] or [Heath, 2006].

Other Known Uses

This pattern is used in many Symbian OS services and a couple of examples are given below. However, it is more common for Symbian OS services to use alternatives to this pattern because they need to mediate access to shared resources. Often this is not obvious, because the service interface supplies a convenience DLL in order to simplify the inter-process communication. In some cases it's not obvious that there is a shared resource, for example in the case where a service (e.g. cryptographic algorithms) may, on some devices, be implemented in hardware.

- *EZLib Compression and Decompression*
EZLib is a CTS provided by Symbian for compressing and decompressing memory streams, files, and file archives. It is packaged as two executables, `ezlib.dll` and `ezip.dll`, which export a C++ interface, `CEZCompressor`, etc., to the open source executable, `libz.dll`, which exports a C interface.
- *DBMS*
This pattern is used by DBMS in order to provide a lightweight service to allow access by a single client to a specific database. This co-exists with a heavier-weight implementation of DBMS based on *Client-Server* (see page 182), which must be used when a database needs to be accessed by multiple clients or when there are restrictions on which processes may use the database.

Variants and Extensions

- *Dynamically Selected CTS*
The Symbian OS ECom service allows a client to select and load a DLL, and hence a CTS, at run time. This gives a client more flexibility than in the main pattern where the client selects a CTS during development. One benefit of this is that the client can cope gracefully with a CTS not being available at run time instead of simply failing to load.
Plug-in selection is done by using the ECom service which is implemented as a *Client-Server* (see page 182) and hence involves a context switch when the plug-in is loaded. However once the ECom plug-in has been selected, it acts just as if the DLL providing the plug-in had been loaded directly into the process; thereafter access has the same performance benefits as statically loaded DLLs.
- *Compiling the CTS Directly into Client Code*
This variant incorporates the actual objects in the static LIB into the calling executable when it is compiled. As a result, every executable that uses the service contains a copy of the code (see Figure 6.7).

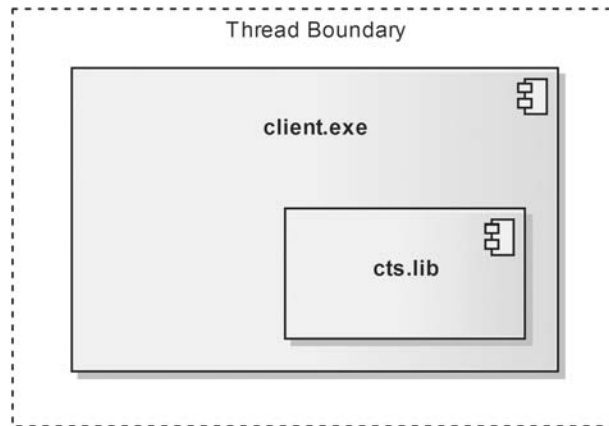


Figure 6.7 Structure of the static LIB client-thread service

There are few (if any) examples within Symbian OS itself of services provided as static LIBs. This is because device creation engineers are rarely in the situation where the benefits of using static LIBs outweigh the costs of the additional code size when compared to using DLLs. Third-party developers, however, sometimes distribute their components as static LIBs in order to defer the responsibility for getting certain Platform Security capabilities to their clients, to remove issues of binary compatibility, and so on.

Services based on this variant cannot be upgraded independently of the client but, on the other hand, only one copy needs to be maintained for all clients (no copying and pasting) and there are no issues of binary compatibility between versions.

This variant is very responsive. As objects are directly included in the calling executable there is no thread or process context switch; indeed there isn't even the, usually very small, start-up cost of loading a DLL.

A further benefit is that since the objects in the LIB are part of the calling executable, they do not need to be assigned capabilities to be used by its clients. This gets around a very common problem for developers of shared components – the need to gain manufacturer capabilities for shared DLLs, even though the DLL itself does not use them.

This variant is very simple to implement. The service provider exports a header file (as for the DLL variant) and supplies a static LIB which contains objects that are incorporated into the calling executable at build time.

- The service provider MMP file specifies the LIB TARGETTYPE:

```
TARGETTYPE lib
```

- The MMP file does not need to specify any capabilities, UIDs, SECUREID or VENDORID, as all of these are inherited from the EXE file used to create the process within which the LIB executes at run time.
- The client MMP file specifies the LIB files using the STATIC-LIBRARY keyword followed by the list of LIBs (both filename and extension are specified):

```
STATICLIBRARY mylib.lib otherlib.lib newlib.lib
```

References

- Chapter 7 explains why a service provided by a DLL cannot securely verify a client's identity or its capabilities.
- *Client-Server* (see page 182) is an alternative to this pattern where the service provider is based in another thread. It provides the necessary infrastructure for sharing and serializing access to resources as well as for allowing security checks on clients.
- *Buckle* (see page 252) discusses the security issues related to loading DLL plug-ins; it is closely related to the *Dynamically Selected CTS* variant of this pattern.
- The following documents provide additional information related to building a CTS:
 - Symbian Developer Library » Symbian OS Tools And Utilities » Build tools guide » How to build DLLs
 - Symbian Developer Library » Examples » Symbian OS fundamentals example code » CreateStaticDLL and UseStaticDLL: using a statically linked DLL
 - Symbian Developer Library » Symbian OS guide » Essential idioms » Exporting and Importing Classes in Symbian OS

Client–Server

Intent Synchronize and securely police access to a shared resource or service from multiple client processes by using the Symbian OS client–server framework.

AKA None known

Problem

Context

Resources and services need to be shared among multiple clients running in different processes.

Summary

Many applications are designed around concurrent objects running in different processes to improve quality of service such as performance and reliability. Often these different objects need to access a shared resource. In this concurrent environment, it is necessary that:

- We have a transparent means of inter-process communication.
- Access to shared resources is synchronized.
- Both synchronous and asynchronous requests from clients can be met easily.
- Where applicable, the access to shared resources, or sensitive data, is policed.

Description

- *Inter-process communication should be transparent.*
The need to protect against memory corruption across process boundaries means that in most cases objects cannot be called passively across this boundary. Such calls must be serialized and passed across the boundary using privileged code. It is impractical for each application developer to do this. So what is needed is an inter-process communication mechanism that provides user-friendly support for transforming the memory representation of an object to or from a data format suitable for storage or transmission, a process known as *marshaling*¹² and *un-marshaling*.

¹²[en.wikipedia.org/wiki/Marshalling_\(computer_science\)](http://en.wikipedia.org/wiki/Marshalling_(computer_science)). See also serializing.

- *Synchronized¹³ access to shared resources should be simple.*
Here, synchronization means the coordination of simultaneous threads or processes to complete a task by correctly ordering them at run time and avoiding unexpected race conditions. Achieving synchronous access to a resource in a collaboration of concurrent objects is not easy using low-level synchronization mechanisms, such as acquiring and releasing mutual exclusion (mutex) locks. In general, service calls that are subject to synchronization constraints should be serialized transparently when an object is accessed by multiple client threads. A common mechanism for doing this is needed.
- *Both synchronous and asynchronous requests from clients can be met easily.*
The actions offered by a service may take any amount of time, from negligible to considerable, even excluding the context switching and message transfer time. Where the time is negligible, clients may make a synchronous request on the service that blocks their current thread of execution to simplify their design. Where the time is considerable, an asynchronous request may be needed to guarantee some quality of service – such as co-operative multitasking in the client's own thread. It is necessary that these different requests can be made very simply by developers of the client code.
- *Access to shared resources and sensitive data is policed.*
It is often the case that you may wish to restrict access to the shared resource or sensitive data from some clients. Such policing necessarily needs to be done close to the service and in privileged code where the developer of the client code cannot tamper with, or bypass, a security policy check.

Example

One example of this problem is the file system. Symbian OS has a microkernel architecture [Buschmann *et al.*, 1996], so its file system isn't part of the kernel facilities. Instead, it manages one or more storage media that are used by a large number of client applications. It needs to be able to control the access to each individual file by a number of clients to guarantee its integrity.

The file system also polices the data cages required by the Platform Security architecture so that only trusted clients with specific security tokens, such as a capability or a SID, may access the restricted directories `/sys`, `/private` or `/resource`. For instance, a client must have the TCB capability to be allowed write access to `/sys`.

¹³ en.wikipedia.org/wiki/Synchronization.

Since both security checks on clients and a way of mediating access to a shared resource are required, *Client-Thread Service* (see page 171) doesn't provide a solution to this problem.

Solution

The solution to this problem is to have the resource managed by a server running in its own process. Clients communicate with the server using messages to request operations on the resource. The communication is bound by a strict protocol that the server defines and is mediated by the Kernel.

As a minimum, this pattern builds on the following Symbian OS patterns with which you should be familiar before continuing:

- *Active Objects* (see page 133) is used to efficiently handle service requests from multiple clients.
- *Request Completion* (see page 104) is used by the server to inform clients of when their requests have been completed.
- *Escalate Errors* (see page 32) is extended so that errors which occur in the server when handling a client's request are passed back to the client.

To help you understand this solution, we describe it in terms of a server that shares a `TInt` with its clients and offers the following actions:

- Get the value synchronously.
- Request notifications of when the value changes.
- Ask the server to perform a long-running activity that updates the value. This request is completed only when the entire activity has completed.

Structure

Here we describe a model of the static relationship between the elements in this architectural pattern. We also define the core responsibility of each element with regards to this pattern.

In this pattern (see Figure 6.8), a server provides some shared service. Clients use a Proxy which conveniently serves as the interface to the server. The connection between a particular Proxy object and the server constitutes a session. A server session object represents the server-side end-point of this connection and may be used to encapsulate any specific information that a server cares to maintain about the client. The client end-point of the connection is encapsulated by the client session.

The client session marshals client requests into request messages that are queued within the server. The server uses *Active Objects* (see

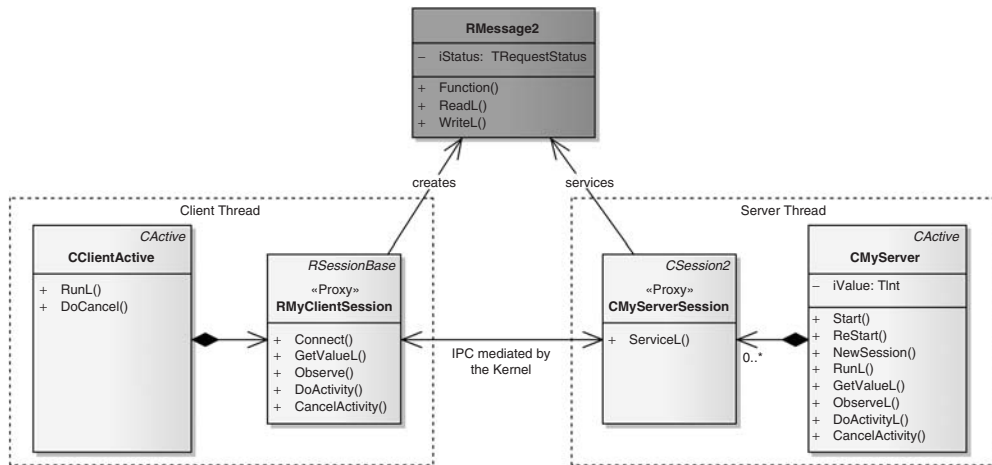


Figure 6.8 Structure of the *Client-Server* pattern

page 133) to schedule the handling of the request messages. When the requests have been serviced, the client is notified via *Request Completion* (see page 104). These participating objects are described in more detail below:

- *Client (CClientActive)*: As the user of a service, the client connects to the server and initiates requests to the server. It waits for and receives a response for each request sent to the server. A client is typically an active object so that it can handle any asynchronous responses from the server without polling or waiting for the request.
- *Server (CMyServer)*: The server encapsulates the shared resource, or sensitive data, to be managed. In effect, it is a *Singleton*¹⁴ from the perspective of the whole software system. It accepts connections from one or more clients. It waits for a request from a client and services it when it arrives. When the client's request is completed, it sends a response. The server object is driven as an active object by the Symbian OS client-server framework which manages the queue of messages¹⁵ waiting for it and feeds the server one request at a time.
- *Client Session (RMyClientSession)*: This session provides the interface that allows clients to invoke the publicly accessible methods on a server. The client session, when required, runs in the same thread as the client and acts as a Proxy to the server. It is employed where the client and server are separated into different execution, or memory management, contexts. It represents and encapsulates the client's

¹⁴This pattern is one way in which *Singleton* (see page 346) is implemented on Symbian OS.

¹⁵This serves as the synchronization point, sequencing messages from different clients to the server so they can be executed in order.

end-point of a session between the server and this particular client. It is responsible for marshaling client calls into a request message that are sent to the message queue in the server context.

- *Request Message (RMessage2)*: The request message allows *Request Completion* (see page 104) to be extended, not only to allow a *TRequestStatus* to be passed between the client and server but also to pass any context information about a specific method invocation on a client session, such as the method parameters.
- *Server Session (CMyServerSession)*: This session represents and encapsulates the server end-point of the connection between the server and a single client. It is responsible for de-marshaling a request message and making the equivalent call on the server. It is further used to manage other client-specific information the server may need.

Dynamics

Here we discuss the typical use cases that are realized in this pattern. The behaviors we consider are:

- starting and connecting to a server
- making a simple synchronous service request
- making an asynchronous service request
- observing state changes in the service
- canceling a long-running service request
- disconnecting from the server.

A number of simplifications are made in the diagrams so that the detail of how the client–server framework provided by Symbian OS operates doesn't get in the way of your understanding of how an implementation of this pattern should behave. In particular, the following things are technically incorrect but conceptually helpful:

- The client–server framework behaves as an active scheduler for both the client and the server process.
- The request message, *iMsg*, sent between the client and the server appears to be the same instance.

Starting and Connecting to a Server

Figure 6.9 describes the interaction between the objects involved in loading and connecting to a server.

Note that your server object must provide a name for itself when it is first started. This name is then used by clients to identify to the

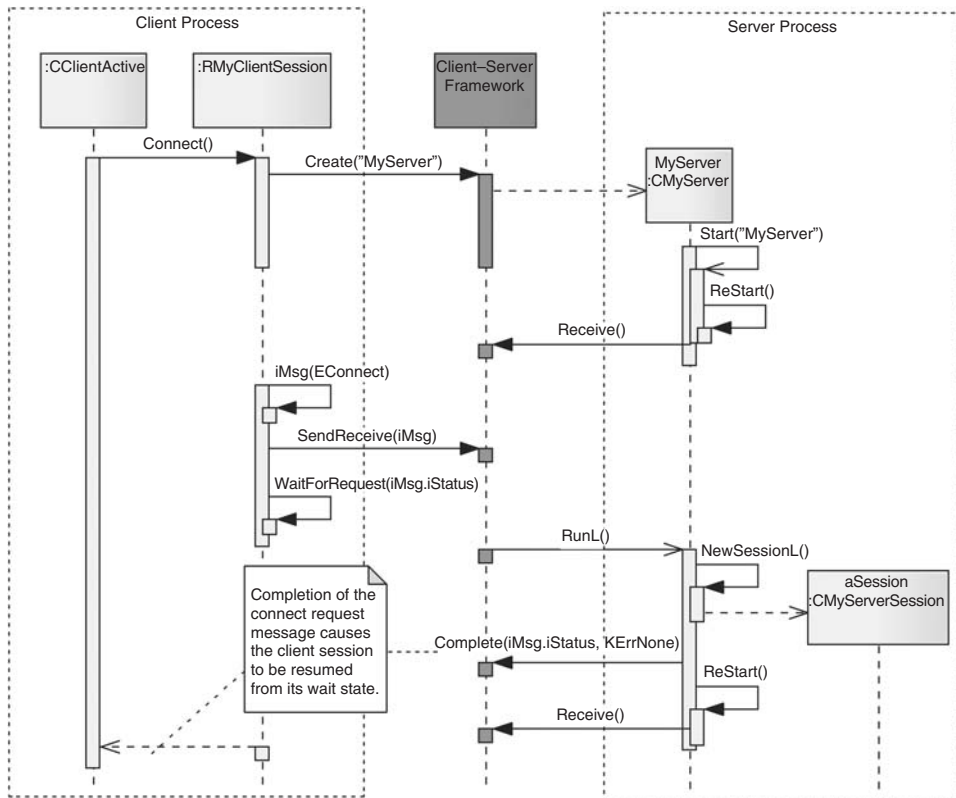


Figure 6.9 Dynamics of the *Client-Server* pattern (first connection to a server)

client-server framework the server to which they wish to connect. This name must be unique across the whole system and hence the `Start()` call fails if another server with the same name already exists.

To prevent denial-of-service attacks from malware deliberately creating servers using your chosen server name, and hence provoking a `KErrAlreadyExists` error when your server later comes to be created, you have the option of prepending a `!` to the front of your server name. To prevent misuse, your server process must have the `ProtServ` capability for this to succeed.

Making a Simple Synchronous Service Request

Figure 6.10 depicts what happens when a client makes a simple synchronous service request on a server.

Making an Asynchronous Service Request

From the server's perspective, all requests are made asynchronously. However, as we see in Figure 6.10, the client session blocks on the

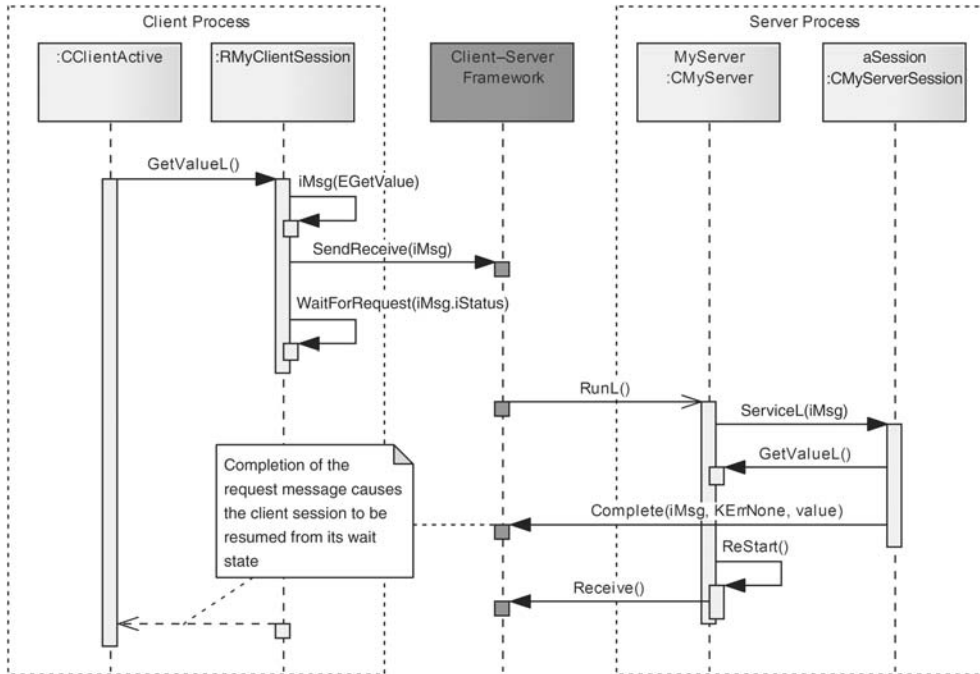


Figure 6.10 Dynamics of the *Client-Server* pattern (simple synchronous request)

`TRequestStatus` it supplies when making what appears to the client as a synchronous call. To make an asynchronous service request on a server, a client must supply its own dedicated `TRequestStatus` to be completed by the server. After the call has been made, the client can choose to do one of two things:

- When using a simple `TRequestStatus`, the client can wait on the `TRequestStatus`, thereby blocking its thread until the request is completed.
- When using the `TRequestStatus` from an active object, the client can return execution control from its current operation. When the request is completed, the active object is scheduled to run and hence notifies the client of the event.

The second of these tactics is the most used as it allows the client's thread to perform other tasks in parallel to the request being serviced by the server. This is the tactic illustrated in Figure 6.11.

Notice the way the server delegates the actual activity to a servant active object. It is a design strategy that frees the server to respond to other messages to maintain quality of service on behalf of all clients. The servant makes use of the 'Long Running Active Object' variant of *Asynchronous Controller* (see page 148) to break up what would otherwise be a long

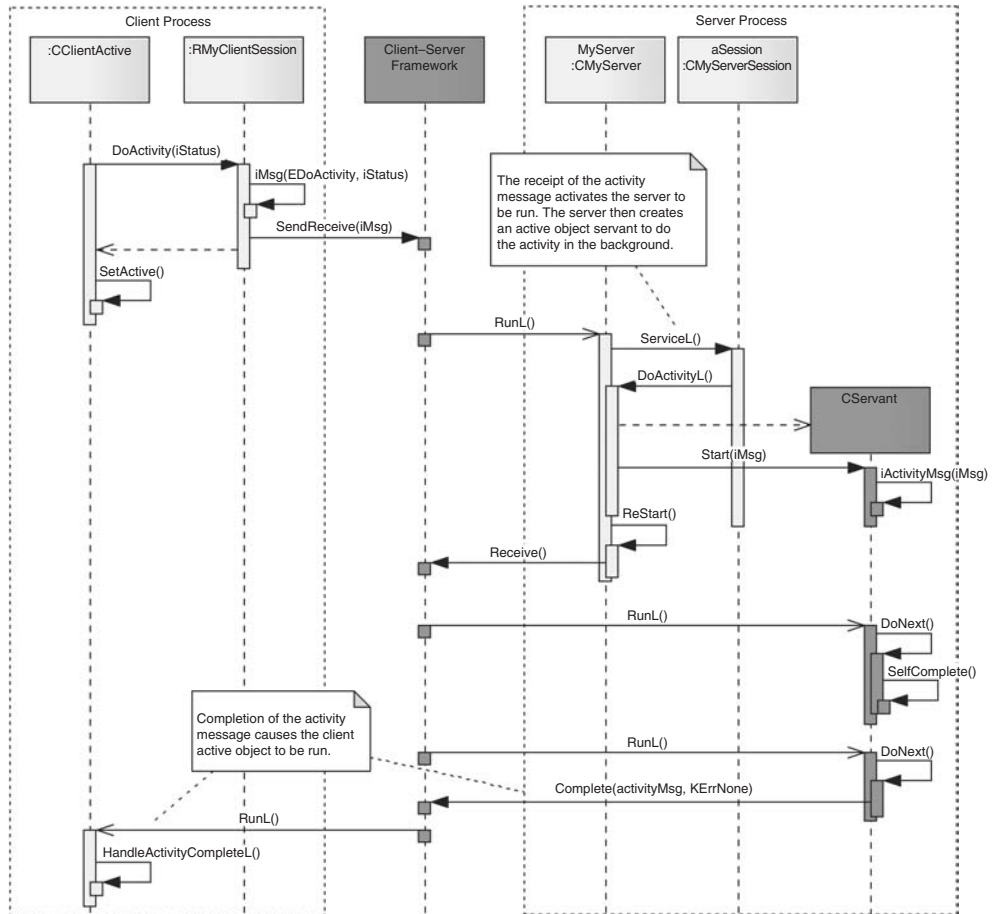


Figure 6.11 Dynamics of the *Client-Server* pattern (long-running asynchronous request)

synchronous activity that would disrupt the entire server thread. The use of *Asynchronous Controller* regularly yields control back to the active scheduler so that higher-priority active objects get a chance to run.

Canceling an Asynchronous Service Request

There are a number of scenarios in which a client may wish to cancel an asynchronous service request that it has made. This pattern must be able to handle all these situations successfully. We summarize them here as:

- Scenario 1: The service request hasn't even begun to execute (but the client does not know that at the point of requesting a cancellation).
- Scenario 2: The service request is being dealt with by a servant active object in the server context when the cancellation request is received.

- Scenario 3: The service request has already been completed and replied to (but the client does not know that at the point of requesting a cancellation).

In all situations, the important thing is that a `TRequestStatus` is only completed once for the service request with which it is associated. To achieve this, the cancel use case is realized synchronously by sending a specific cancellation request to the server with its own separate `TRequestStatus`.

There are two approaches corresponding to the choices the client made when making the asynchronous request:

- If a simple `TRequestStatus` is used to make the activity request, then the cancel request is made before the client blocks the thread to wait on the same activity `TRequestStatus`.
- If an active object is used to make the activity request, then cancellation is achieved by canceling the active object which then waits on its `TRequestStatus`.

In Scenario 1, the client session may have failed to create the resources needed to send the original request message. Alternatively, the server session may have received the activity request but failed to execute it. Either object must have already completed the original `TRequestStatus` with a failure code. So the server has no pending activity to cancel. The server simply acknowledges the cancel request by completing it with a success condition code (`KErrNone`).

In Scenario 2, the server must quickly abort the servicing of the original activity request and complete it with a `KErrCancel`. The server also acknowledges the cancel request with a `KErrNone`. Since the cancellation is being realized synchronously it is important that the server response is immediate. So any expensive (in time) pre-cessation tasks should be deferred until after the `TRequestStatus` is completed.

In Scenario 3, the server must simply acknowledge the cancel request message with `KErrNone`. There is no pending activity request message to be canceled.

Figure 6.12 depicts Scenario 2. It shows a server that has just created a servant active object to carry out the activity when the cancellation notice arrives. Notice how the client active object waits on its `TRequestStatus` until it is completed, in this case, with `KErrCancel`.

Observing a Server's State

Making a call to observe a server's state is similar to making any other asynchronous request for information. Of course one difference is that the client may only require to be notified of the state change instead of receiving the changed information. In the second instance, the client must provide a placeholder for the information.

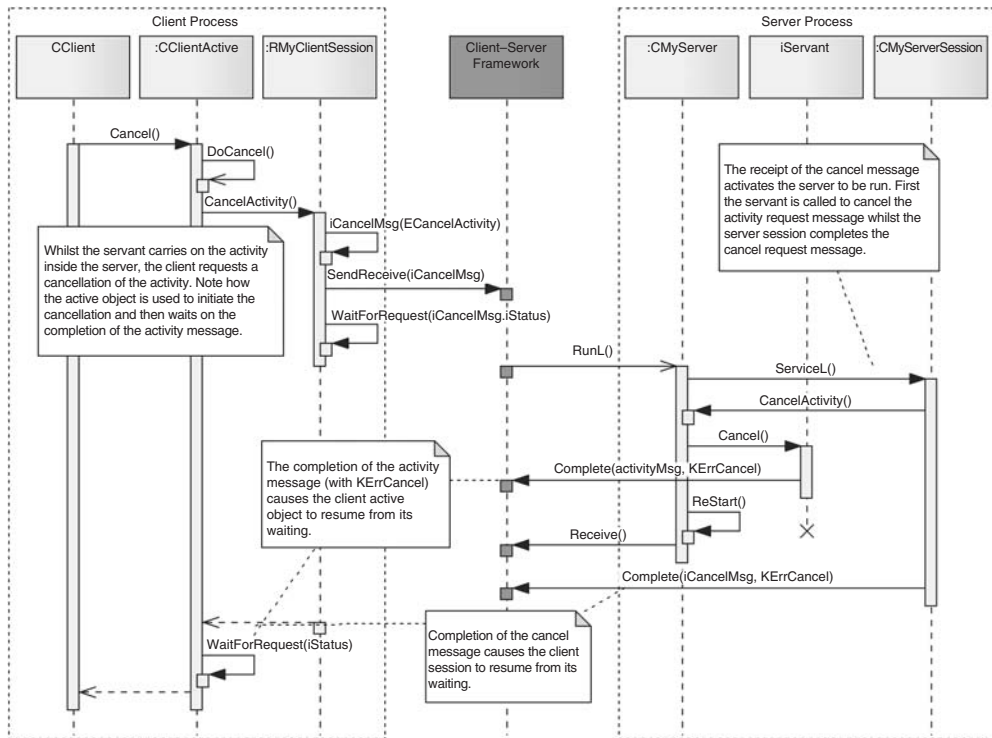


Figure 6.12 Dynamics of the *Client-Server* pattern (cancellation of a pending asynchronous request)

Disconnecting from a Server

Disconnection from the server is done in the same way as making any other synchronous request. The client session sends an `EDisconnect` message to the server. The server intercepts this message, instead of dispatching it to the server session, to close and destroy the server session. If this is the last session closed then a transient server also un-registers and destroys itself. In effect, this is an application of *Lazy De-allocation* (see page 73) where the server is the resource.

Note that the Symbian OS client-server framework ensures that the server is kept notified of which clients are connected to the server at all times. This is true even if the client thread is terminated, such as if it panics, as an `EDisconnect` message is still sent to the server. This is often a key distinction between this pattern and other ways of implementing IPC.

Implementation

This pattern is codified into the client-server architectural framework of Symbian OS. You will find a lot of information describing the framework

in books such as [Babin, 2007], [Harrison and Shackman, 2007] and [Heath, 2006] as well as in the Symbian Developer Library. For the complete implementation of this pattern, please see those sources of information.

Here we provide code snippets to illustrate how to implement a server and a client session based on the existing Symbian OS client-server framework. We also describe code showing how a typical client interacts with the server through its client session.

Important Note: The code snippets are provided for illustration purposes only. To keep the information manageable we don't always include the typical house-keeping code needed to avoid resource leaks and so on.

Implementing the Server Side

- *Creating and starting the server*

The system loads the server's code¹⁶ when it is required for the first time. It passes control to `E32Main()` which is responsible for constructing and running the server object. Note that, for clarity, the implementation below shows a non-transient server:

```
// CMyServer
inline CMyServer::CMyServer()
    :CServer2(CActive::EPriorityStandard, ESharableSessions)
{}

CServer2* CMyServer::NewLC()
{
    CMyServer* self = new(ELeave) CMyServer;
    CleanupStack::PushL(self);
    self->ConstructL();
    return self;
}

// 2nd phase construction - ensure the server object is running
void CMyServer::ConstructL()
{
    StartL(KMyServerName);
}

// CMyServerSession
inline CMyServerSession::CMyServerSession()
{}

inline CMyServer& CMyServerSession::Server()
{
    return *static_cast<CMyServer*>(
        const_cast<CServer2*>(CMyServerSession2::Server()));
}
```

¹⁶Usually packaged within an EXE file.


```

// Perform all server initialization, in particular creation of the
// scheduler and server before starting the scheduler
static void RunServerL()
{
    // Naming the thread after the server helps to debug panics
    User::LeaveIfError(RThread::RenameMe(KMyServerName));

    // Create and install the server active scheduler
    CActiveScheduler* s = new(ELeave) CActiveScheduler;
    CleanupStack::PushL(s);
    CActiveScheduler::Install(s);

    // Create the server (leave it on the cleanup stack)
    CMyServer::NewLC();

    // Initialization complete, now signal the client
    RProcess::Rendezvous(KErrNone);

    // Ready to run
    CActiveScheduler::Start(); // Doesn't return until the server dies

    // Clean up the server and scheduler
    CleanupStack::PopAndDestroy(2);
}

// Entry point to server process
TInt E32Main()
{
    __UHEAP_MARK;
    CTrapCleanup* cleanup = CTrapCleanup::New();
    TInt r = KErrNoMemory;
    if (cleanup)
    {
        TRAP(r, RunServerL());
        delete cleanup;
    }

    __UHEAP_MARKEND;
    return r;
}

```

- *Responding to a client connection request*

A client's connection request is intercepted by the constructed server in its base implementation, CServer2. At some point, the virtual function NewSessionL() is called on the sub-class to construct a specific server session object to manage the connection to the client.

```

// Create a new client session.
CMyServerSession2* CMyServer::NewSessionL(const TVersion&,
                                           const RMessage2&) const
{
    // Could also check that the version passed from the client-side
    // library is compatible with the server's own version.

```

```

    return new(ELeave) CMyServerSession();
}

// Called by the framework when a new session is created
void CMyServer::AddSession()
{
    ++iSessionCount;
}

```

- *The server message dispatch*

The server dispatches client messages to the session to un-marshall and handle. In the example below, the session calls specific methods on the server corresponding to the request. By going through the session first, the framework allows the developer to do any client-specific pre-processing of the message in the session before it calls the server.

```

// Handle a client request
void CMyServerSession::ServiceL(const RMessage2& aMessage)
{
    switch (aMessage.Function())
    {
    case EGetValue:
        GetValueL(aMessage);
        break;
    case EDoActivity:
        iActivityMsg = aMessage;
        Server().DoActivity(*this);
        break;
    case EObserve:
        iNotifyMsg = aMessage;
        Server().ObserveL(*this);
        break;
    case ECancelActivity:
        Server().CancelActivity();
        aMessage.Complete(KErrNone);
        break;
    default:
        aMessage.Panic(KServerPanic, EPanicIllegalFunction);
        break;
    }
}

```

- *Server error handling*

In servicing the message, the server session relies heavily on *Escalate Errors* (see page 32) this is partly because it reduces the amount of error-handling code but mainly because errors are escalated to a point at which there is sufficient context to handle them effectively. The first opportunity to resolve any errors is in this pattern's equivalent of `RunError()` which is also optional:

```
void CMySession::ServiceError(const RMessage2& aMessage, TInt aError)
{
    if (aError == KErrBadDescriptor)
    {
        // A bad descriptor error implies a badly programmed client, so
        // panic it
        aMessage.Panic(KServerPanic, EPanicBadDescriptor);
    }

    // Otherwise escalate the error to the client-server framework
    CSession2::ServiceError(aMessage, aError);
}
```

Normally an error can only be escalated within a single thread, however `CSession2::ServiceError()` will handle errors that reach it by completing the message currently being serviced with the error. This allows you to escalate an error back into the client's thread¹⁷ where it is often much more appropriate to handle the error.

An alternative error-handling strategy is to use *Fail Fast* (see page 17) to panic programming faults. Just like *Leave*, a panic normally affects just the local thread. However, the client-server framework gives you the option of panicking a client if you've detected that it is at fault. This is done by calling `Panic()` on the `RMessage2` object you're currently servicing; the client the message came from will be terminated. In the `ServiceL()` implementation, if an unknown message type is received then someone must be sending messages directly without using the client session which, for released code, is probably a sign of an attempted security attack.

- *Providing a simple request service*

In the server message dispatch code snippet above you'll notice how `CMyServerSession` interprets the `EGetValue` message into a call on `CMyServer`:

```
void CMyServerSession::GetValueL(RMessage2& aMessage)
{
    TInt results = Server().GetValue();
    TPtr ptr((TUint8*)&results, sizeof(results), sizeof(results));
    aMessage.WriteL(0, ptr);
    aMessage.Complete(KErrNone);
}

TInt CMyServer::GetValue()
{
    return iValue;
}
```

¹⁷In the client session, you need to *Leave* again if you wish to escalate the error up the client's call stack.

- *Observing changes to the server value*

The example below shows a server that only allows one client at a time to observe it. Here the observer is the server session. This abstraction is done to make the server code more portable – it is able to collaborate with clients in different configurations, for example in a unit test environment. For simplicity, the notification style here doesn't include the new value so clients would have to make an `RMyClientSession::GetValueL()` call to retrieve it if needed.

```
void CMyServer::ObserveL(MObserver& aObserver)
{
    iObserver = &aObserver; // Supports just one observer, for the sake
                           // of brevity
}

TInt CMyServer::ChangeValue()
{
    ++iValue;
    if (iObserver)
    {
        iObserver->NotifyChange(KErrNone);
        iObserver = NULL;
    }
    return iValue;
}

void CMyServerSession::NotifyChange(TInt aError)
{
    iNotifyMsg.Complete(aError);
}
```

- *Initiating an asynchronous service*

`CMyServerSession` calls `CMyServer::DoActivityL()` in response to the `EDoActivityL` command. The server delegates the activity to a servant active object, `CMyServant`. `CMyServant` is designed as an *Asynchronous Controller* (see page 148) to break down its task into smaller steps so it can run in the server's thread.

```
void CMyServer::DoActivityL(MActivityEventMixin& aActivityEventMixin)
{
    // This server is only designed to have one of this
    // activity running at a time;
    CancelActivity();

    iServant = CMyServant::NewL(aMixin);
    iActivityEventMixin = &aActivityEventMixin;
    iValue = 0; // Reset the value at the start of the activity
}
```

- *Performing the asynchronous activity and notifying change*
The servant performs the delegated activity in small steps. In this example, each step calls the server to change its value and to notify clients of the change.

```
const TInt KMaxCount = 100000;

// from CActive
void CMyServant::RunL()
{
    NextStep();
}

void CMyServant::NextStep()
{
    // Do some activity that changes the server value
    TInt current_value = iServer.ChangeValue();
    if (current_value == KMaxCount)
    {
        iServer.ActivityComplete();
    }
    else
    {
        // Activate self to be run again
        SelfComplete();
    }
}
```

- *Completing the activity*
When the servant has completed its work, it calls `CMyServer::ActivityComplete()`. The server then notifies the server session of the completion.

```
void CMyServer::ActivityComplete()
{
    // The servant has completed its work so dismiss it
    delete iServant;
    iServant = NULL;

    if (iActivityEventMixin)
    {
        // Inform the client that the work is completed
        iActivityEventMixin->ActivityComplete(KErrNone);
        iActivityEventMixin = NULL;
    }

    if (iObserver)
    {
        // There's going to be no more changes so
        // notify the observer to ensure it doesn't wait in vain
    }
}
```

```

        iObserver->NotifyChange(KErrNone);
        iObserver = NULL;
    }
}

void CMyServerSession::ActivityComplete(TInt aError)
{
    iActivityMsg.Complete(aError);
}

```

- *Canceling the activity*

If the activity is canceled early then the server needs to notify the server session of the completion:

```

void CMyServer::CancelActivity()
{
    if (iServant)
    {
        // Dismiss the servant
        delete iServant;
        iServant = NULL;
    }
    if (iActivityEventMixin)
    {
        // Inform our client that the work is completed
        iActivityEventMixin->ActivityComplete(KErrCancel);
        iActivityEventMixin = NULL;
    }
    if (iObserver)
    {
        // There's going to be no more changes so
        // notify the observer to ensure it doesn't wait in vain
        iObserver->NotifyChange(KErrCancel);
        iObserver = NULL;
    }
}

```

Implementing a Client Session – RMyClientSession

The client session should be packaged into a client-side DLL separate from `server.exe`. The client-side DLL should not depend on `server.exe` so that the server is decoupled from the client and, in particular, avoids forcing clients to have all the static dependencies of the server. However, since the client-side DLL will be relatively small, it is normally not a problem for `server.exe` to depend on the client-side DLL, such as for any types that need to be shared between the client and the server.

```

// Number of message slots available per session.
const TInt KMyServerDefaultMessageSlots = 1;
const TInt KMyServerRetryCount = 2;

```

```

// Start the server process. Simultaneous launching of two such
// processes should be detected when the second one attempts to
// create the server object, failing with KErrAlreadyExists
static TInt StartServer()
{
    const TUidType serverUid(KNullUid, KNullUid, KServerUid3);
    RProcess server;
    TInt r=server.Create(KMyServerImg, KNullDesC, serverUid);
    if (r != KErrNone)
    {
        return r;
    }
    TRequestStatus stat;
    server.Rendezvous(stat);
    if (stat != KRequestPending)
    {
        server.Kill(0); // Abort startup
    }
    else
    {
        server.Resume(); // Logon OK - start the server
    }
    User::WaitForRequest(stat); // Wait for start or death
    // We can't use the 'exit reason' if the server panicked as this
    // is the panic 'reason' and may be '0' which cannot be
    // distinguished from KErrNone
    r = (server.ExitType() == EExitPanic) ? KErrGeneral : stat.Int();
    server.Close();
    return r;
}

// Connect to the server, attempting to start it if necessary
EXPORT_C TInt RMyClientSession::Connect()
{
    TInt retry = KMyServerRetryCount;
    for (;;)
    {
        TInt r = CreateSession(KMyServerName,
                               TVersion(0,0,0),
                               KMyServerDefaultMessageSlots);
        if (r != KErrNotFound && rKServer != KErrServerTerminated)
        {
            return r;
        }
        if (--retry == 0)
        {
            return r;
        }
        r=StartServer();
        if (r != KErrNone && r != KErrAlreadyExists)
        {
            return r;
        }
    }
}

EXPORT_C TInt RMyClientSession::GetValueL()

```

```

    {
        TPkgBuf<TInt> pkg;
        TIpcArgs args(&pkg);
        User::LeaveIfError(SendReceive(EGetValue, args)); // Synchronous
        return pkg();
    }

EXPORT_C void RMyClientSession::Observe(TRequestStatus& aStatus)
{
    // Note that the asynchronous version of SendReceive doesn't return
    // any errors because they all go through aStatus
    SendReceive(EObserve, aStatus);
}

EXPORT_C void RMyClientSession::DoActivity(TRequestStatus& aStatus)
{
    SendReceive(EDoActivityL, aStatus);
}

EXPORT_C void RMyClientSession::CancelActivity()
{
    SendReceive(ESendCancelActivity);
}

```

Using the Server

To illustrate how a client might work, we use a single class `CClient` to manage two active objects, `CObserver` and `CActivity`, that use the two asynchronous actions provided by the server:

```

void CClient::UseServerServicesL()
{
    // Here we make a simple synchronous request
    iServerValue = iMyClientSession.GetValueL();

    // We now call the server to perform some long-running activity
    // that continuously changes its value

    // We want to observe the server for changes to its value so we create
    // an active object to listen for the change notice
    iObserver = CObserver::NewL(iMyClientSession);

    // First we create an active object that will listen for the
    // completion notice. We pass in a reference to the observer to tell
    // it when to stop
    iActivity = CActivity::NewL(iMyClientSession, &iObserver);

    // Now we make the actual service requests
    iActivity->DoActivity();
    iObserver->Observe();
}

void CActivity::DoActivity()
{
    {

```



```

        iMyClientSession.DoActivity(iStatus);
        SetActive();
    }

void CObserver::Observe()
{
    iMyClientSession.Observe(iStatus);
    SetActive();
}

```

The client then just has to implement `CActivity::RunL()` and `CObserver::RunL()` to handle the completion of the requests. For instance:

```

void CActivity::RunL()
{
    iObserver.Cancel();
}

void CObserver::RunL()
{
    User::LeaveIfError(iStatus);

    // Make the observation request again
    Observe();

    iServerValue = iMyClientSession.GetValueL();
    ... // Do something with the value
}

```

Consequences

Positives

- *Resource sharing* – resources will always remain scarce on a mobile device so the need to share will always be there. This pattern helps developers to meet the need wherever it arises.
- *Protected resource* – all the data is stored with the server, which generally has far greater security controls than most clients. A server in its own process also means its data is protected from misbehaving clients. Servers can better control access to resources to guarantee that only those clients with the appropriate permissions may use the resource. Centralized data also means that updates to the data are far easier to administer than would be possible if it were distributed.
- *Support for clients with diverse abilities and behavior* – the physical separation of the server from its clients means it is able to work with multiple clients of different abilities in the operating system. It is also easier to design how such collaboration will work.

- *Serialized access* – this pattern removes the burden of synchronizing access to a shared resource from the developers of each individual client.
- *Decoupled services* – the mature client–server architecture implementation of Symbian OS provides a usable API and infrastructure for realizing other smaller patterns such as *Active Objects* (see page 133) and *Cradle* (see page 273).

Negatives

- *Latency* – the physical separation of client and server means communication always requires a context switch, taking in the order of microseconds,¹⁸ that adds to the latency of the service and reduces performance.
- *Congestion* – as the number of clients increases so does the load on the server and the risk to quality of service guarantees. For instance, the server may no longer be able to meet performance targets for critical use cases because it is busy with a previous less critical use case. It may also become bogged down with notifying other clients of changes that one client makes.
- *Reduced system robustness* – the failure of a server may mean the failure of all its dependent clients which introduces a single point of failure into the system.
- *Complexity* – the separation between clients and servers introduces complex indirection and concurrency situations that have to be fully appreciated by developers. This makes testing and debugging more difficult than with a passive relationship. Black-box testing of the functionality of a server through a single client using the server's client session may be fairly straightforward. At the system level, however, it is difficult to sufficiently test all the scenarios that multiple-client engagement with the server entails. Developers of a server also face a real challenge of designing and implementing a server whose elements can be sufficiently unit-tested in a directly coupled configuration. Finally, debugging is more complex. Tracing through a typical collaboration between a client and a server can involve so much context switching as to make tracking progress difficult.
- *Increased resource usage* – By default the extra process required by this pattern will consume a minimum of 21–34 KB¹⁸ of RAM; code size is increased as a result of the complexity of the solution, and the

¹⁸See Appendix A for more details.

multiple objects required in the client, the server and the kernel¹⁹ all consume RAM of their own.

See the variants and extensions of this pattern for other architectural quality consequences to the use of this pattern.

Example Resolved

The example problem we chose is the File Server. It is implemented in two main executables:

- `efsrv.dll` provides the client session, RFs, as well as a number of supporting types such as `TVolumeInfo`, `CDir`, etc. Since a client might wish to use multiple files at once, the sub-session²⁰ class, `RFile`, is provided to support this efficiently.
- `efile.exe` provides the executable code used to create the process in which the File Server runs. `efile.exe` depends on `efsrv.dll` to provide the common types used by both the server and its clients.

To avoid duplicating a lot of the information shown above, we just give some examples here of how the client session services are implemented.

The following code shows an example where some validation of the client's arguments is done client-side to avoid a wasted round-trip to the server:

```
EXPORT_C TInt RFs::Rename(const TDesC& anOldName, const TDesC& aNewName)
{
    if (anOldName.Length() <= 0 || aNewName.Length() <= 0)
    {
        return (KErrBadName);
    }
    return(SendReceive(EFsRename, TIpArgs(&anOldName, &aNewName)));
}
```

This next example shows parameters being used to describe the request (`aDrive` is the drive to get information about) as well as to return information (`anInfo` is a complex type containing drive information for `aDrive`):

```
EXPORT_C TInt RFs::Drive(TDriveInfo& anInfo, TInt aDrive) const
{
    TPckg<TDriveInfo> m(anInfo);
    return(SendReceive(EFsDrive, TIpArgs(&m, aDrive)));
}
```

¹⁹The kernel maintains a number of objects to support the client-server framework that mirror those used on the user side such as `CServer2`, `CSession2`, etc.

²⁰See the Variants and Extensions discussion.

See the Symbian Developer Library for example code on how to use the File Server interface.

Other Known Uses

The client–server framework is one of the defining structures of Symbian OS, alongside descriptors and active objects. It provides a concrete implementation of a secure service and hence is at the heart of the operating system’s security strategy. As such, it is not surprising that a large number of technologies in the operating system have been designed as servers. Most of the base and middleware services in the operating system, such as Contacts Management, Calendaring, the Graphical User Interface Engine, the Database Management System, Telephony and Sockets Communication, are all implemented in servers. Application clients interact with these servers through Proxies, some trivial and some very substantive.

Specific examples from the Personal Information Management technology area alone are `alarmserver.exe`, `agnserver.exe` (Calendaring), `contacts.exe`, and `tzserver.exe`.

Variants and Extensions

Here we look specifically at advanced strategies that can be applied to enhance a number of architectural qualities of a client–server implementation. We first give consideration to general points about object location, communication types and styles in this pattern. We then consider other qualities: functionality, security, extensibility and configurability, usability, and performance.

Relative Server Location

The server, as an active object, realistically needs its own thread of execution. On most operating systems, the fundamental unit of logical memory isolation is the process. So for maximum protection for its managed resource, it is usually the case that the server runs in its own process. Doing this also allows it the maximum autonomy for sharing its resources with all other client processes. It is only in this configuration that this pattern is able to provide a secure service by ensuring its memory is isolated from any interference by clients.

Of course where a single application deploys a server for exclusive use of its own client modules then there is no security threat to protect against and hence the server can be hosted in the same process as the application. After all, no other process needs that service. A common reason to host a server in the same process as its client is that it provides one of the easiest mechanisms in Symbian OS to coordinate multithreading within a single process.

The types of partitioning we have described so far are examples of *single-seat* client-server architectures which are realized on the same computer. This is the main concern of this exposition. Nevertheless, it is worth saying a word about client-server architectures in the world of enterprise computing. Here it often means ultimate separation of client and server on physically different computers on a network – whether PAN, LAN or WAN. This environment is epitomized by the web with its web servers and browser clients. In this environment, there are *two-tier* architectures, involving a client and a server, and *multi-tier* architectures where a server is a client to another server. It goes without saying, though, that multi-tiered architectures can also be realized on a single mobile device.

Server Lifetime

Most servers are usually created to serve the needs of one or more clients. So these servers typically run until the last of their sessions with a client closes. These servers are often described as *transient servers* and are an example of using *Lazy De-allocation* (see page 73).

Then there are *Permanent* ('*Daemon*') servers that, once created, continue running to perform system services, even when there are no clients connected, until shut down by the system. A server may also be made to run permanently if responsiveness is a key requirement. Both of these cases are an example of applying *Immortal* (see page 53) to this pattern.

Types of Request and Communication Styles

In a client-server relationship, the client always initiates a dialog by making a request. The request can be one of the following:

- peek (get) some server object's state, or observe its change
- poke (post) some server object's data, or request some action on it.

Each of the request types may be best served synchronously, if communication bandwidth and server response times permit it, or asynchronously otherwise.

Server state change may be observed in two ways:

- *Information Subscription*: the server responds with the new information. This is a *hot communication* relationship. In some implementations, this may require that the client set aside adequate space to receive the information.
- *Change Notice Subscription*: the server is only obliged to respond with a simple 'state has changed' notice but without the actual information. This is a *warm communication* relationship. It gives clients control over when they get the new information but at the cost of making another round-trip communication.

Server state observation, such as in the Observer Pattern [Gamma *et al.*, 1994], poses real performance engineering difficulties to server designers. What does one do when there are a large number of clients to be notified about a lot of changes? We'll consider strategies for handling some of these problems in the performance variants and extensions.

Functionality

Publish and Subscribe (see page 114) supports server state observation by unconnected clients. It provides the solution to clients who do not have a need to use a server except to observe changes in its state. The server developer determines which data items have such an appeal and publishes them. For example, `tzserver.exe` publishes the time zone in this way because it affects the local time and a number of unconnected clients need to adjust their behavior accordingly. This pattern is not a substitute for connected clients observing server state changes. It is better to provide connected clients with specific server observation methods.

Security

Concepts such as Capability Management and Data Caging [Heath, 2006] as well as the memory isolation offered by processes in Symbian OS enable a server to secure its sensitive data and other resources. In addition, the client-server framework provides the `CPolicyServer` class to allow a consistent policy of security checks to be applied to client requests.

Extensibility and Configurability

Cookies and Trusted Extension Components allow for the configuration of client-specific, server-side behavior. Sometimes the basic functionality offered inside a server needs to be configured with a behavior that is unique to a specific client or client type. It may also be that under these circumstances it is unacceptable, if not impossible, to transfer the data to the client to carry out the operation.

A *Trusted Extension Component* is a plug-in which is loaded server-side at run time using a secure plug-in pattern such as *Buckle* (see page 252). It can be used to provide the configuration needed in the server. The use of Extension Components and any other client-specific server configuration can be managed using a *cookie*. A cookie is a unique piece of token information that the server gives to the client as an identity. Whenever the client connects and provides this token, the server recalls and configures the client session with any previously stored information under the token.

Usability

- *APIs for Making Blocking (Synchronous) Calls on the Server*
This simplifies the design of the client code at the expense of denying other services in the client thread from running. In this pattern, all calls on the server are decoupled from their execution using a message queue. So from the server's perspective all requests are asynchronous and a `TRequestStatus` must be provided to be completed at the end of the operation. These asynchronous calls can be made to appear synchronous from a client's perspective if an intermediary blocks and waits on the `TRequestStatus` until it is completed. Such a requirement will be common to all clients and it makes sense if the blocking behavior is provided as a standard behavior of the client session.
- *Indicating Failure during an Asynchronous Call*
There must be only one way for a client to receive the response to an asynchronous call and that is through the `TRequestStatus` passed in by the client. Failure to do so either in the client session or on the server side can lead to stray events. The client-server framework helps with this by ensuring that default Leaves which occur server-side are used to complete the client's request.
- *Extending Client Sessions with Server Behavior*
Sometimes a non-local server's behavior needs to be augmented with client-side behavior to meet some architectural quality goal. We mention a few of these, such as transaction batching and data caching, under the performance variants and extensions. These require non-trivial proxies that do more than just marshal calls to servers. At other times, the behavior realized by the collaboration of server-side objects needs to be replicated client-side. This is especially useful in situations where the server models an application domain of interacting real-world entities and it is good for the service API to explicitly model these relationships. Examples of such substantive and thick proxies include the interface to the Calendar server and the Window server.

Performance

- *Sub-sessions*
If a client wishes to have multiple simultaneous uses of a server then a server should provide sub-sessions. This allows a client to have one session with the server that each of its sub-sessions use to provide the communication channel to the server. This is more efficient than each of the sub-sessions having their own separate channel. An example of this is the File Server, where each opened file is handled through a separate sub-session.

- *Shared Memory*
Inter-process communication involves copying messages from one address space to another. This consumes valuable memory, even if only temporarily, as well as execution time. These costs can be substantial for large messages so, in such cases, the use of shared memory chunks is recommended to avoid copying data unnecessarily. The Symbian OS Font Bitmap Server uses this technique to provide information to the Window Server and clients.
- *Bulk Transaction API*
This is where the server interface, and hence the client session, is designed to allow clients to make requests for information for a number of server objects with a single message. The Contacts Server Synchronization APIs provide the ability to fetch details about a number of contacts in a single transaction. The same approach can be used for requesting actions on a number of objects all at once.
- *Transaction Batching*
In this variant, a client session is designed to batch a number of requests made by a client before sending it to the server as a single transaction. It depends on and extends the Bulk Transaction API variant. The delimiting factors are the batching buffer size and a timeout over which the batching can be done without noticeable user experience degradation. The Symbian OS Window Server uses this technique.

The same technique can be applied to batching a number of server change notices together and notifying the client in a single reply. The Calendaring Server (`agnserver.exe`) uses this technique to notify clients of calendar events that have changed. Since it is the client who initiates all calls, the server has to deal with the client not calling back quick enough to retrieve the full buffer after being notified. If we need to add to the buffer before the client has re-posted their request then we can replace the content with a single 'all- change' notice and stop adding any more until it is fetched.
- *Data Caching*
This is where either the client or the client session is designed to cache a copy of the server's data. The `tzserver.exe` client session holds such a cache containing both a single compressed time zone rule and the latest set of actualized local time changes produced from it. Client requests for time conversions between universal co-ordinated time and the current local time zone are met, when possible, by the client session using its cache. The client session replaces the cache if the client requests a different time zone. It also observes the server for changes other clients make to the selected zone and flushes the cache when that happens.

- *Co-location of Related Services*
Context switching between threads is expensive; context switching between threads in different processes is even more expensive because of MMU²¹ changes and the various caches to be flushed. There is also an extra cost to data transfer between threads in different processes because of the mapping that needs to be done. Where it is appropriate that servers should be co-located in the same process to minimize this cost, it should be done. A group of servers that depend heavily on each other's resources and that can be trusted to collaborate in the same process space is one example. The Communications framework in Symbian OS is modeled after this strategy. Another example is a server that only serves one specific application. The latter server is best located inside its application's process.
- *Deferred Server Termination*
In situations where clients of a transient server come and go frequently, it is important, for performance reasons, that the server deferred its termination until some definite time after the last session closes. This is especially desirable where the cost of creating and initializing the server's resources are very high. *Lazy De-allocation* (see page 73) can be used to address this by using a timer running alongside the server that destroys the server, and itself, after a set timeout following the closure of the last session.
- *Delegate Child Active Objects*
To provide performance guarantees to all its clients, it is necessary for a server to delegate long-running requested activities to child active objects so it is free to handle the next request. Often these children are *Asynchronous Controllers* (see page 148) running in the server's thread. Sometimes it is best that they have their own thread of execution.

References

- Some related, but contrasting, architectural patterns are Peer-to-Peer [Buschmann *et al.*, 1996] and Master-Slave [Buschmann *et al.*, 1996].
- An implementation of the client-server architecture may use standard industry patterns such as:
 - Proxy [Gamma *et al.*, 1994] as an object adaptor of the server
 - Memento [Gamma *et al.*, 1994] to persist calls on the server
 - Pipes [Buschmann *et al.*, 1996] for communication.

²¹ en.wikipedia.org/wiki/Memory_management_unit.

- A number of Symbian OS patterns are also used:
 - *Active Objects* (see page 133) is used to efficiently handle service requests from multiple clients.
 - *Request Completion* (see page 104) is used by the server to inform clients of when their requests have been completed.
 - *Escalate Errors* (see page 32) is extended so that errors which occur in the server when handling a client's request are passed back to the client.
- The client–server framework chapter in [Harrison and Shackman, 2007] gives comprehensive coverage of how the pattern is codified in Symbian OS.
- [Stichbury, 2004] also covers the client–server framework.
- [Heath, 2006] contains a chapter describing how to design a secure server.
- Template code for a transient server is available from ***developer.symbian.com/main/oslibrary/cpp_papers/advanced.jsp***.
- The Symbian Developer Library contains an overview of the Symbian OS client–server architecture as well as example code:
 - Symbian Developer Library » Symbian OS guide » Base » Using User Library (E32) » Inter Process Communication » Client/Server Overview
 - Symbian Developer Library » Examples » User library example code » Client/Server example code.

Coordinator

Intent Co-ordinate state changes across multiple interdependent components by using a staged notification procedure that maximizes flexibility.

AKA State-Aware Components, Staged System Transition

Problem

Context

You need to synchronize a collective change across many related components in different processes whilst respecting their layering.²²

Summary

- You need to ensure that the dependencies between components are respected by having a structured system of notifications.
- You'd like the state change to be completed as quickly as possible.
- To make the testing and maintenance of your software easier you want the components being notified of the state changes to be loosely coupled both with respect to each other and to the service informing them of the change. In particular, it should be easy to change which components are taking part in the notifications.
- A transparent means of inter-process communication (IPC) is needed.

Description

When synchronizing a collective change across multiple components, you're going to need to perform a series of operations. They might be as wide-ranging as needing to notify components in the system that the free system memory is low or relatively focused, such as informing a collection of UI components in an application that a list has been updated. The more components that need to react to such a change increases the likelihood that there are pre-existing dependencies between some of them.

Some dependencies between components can be straightforward, such as starting a new process with no further interactions between the creator and the created processes. More often, you will find that they require more direct cooperation between them. For components in different processes, they will need to use one of the many IPC mechanisms available in Symbian OS. Examples of commonly used mechanisms are *Client-Server*

²²As in the Layers pattern [Buschmann *et al.*, 1996].

(see page 182) and *Publish and Subscribe* (see page 114). However, regardless of the method used, each sets up a dependency between the two components as a result of the communication. Sometimes these dependencies might be obvious, such as with `RASCLISession` where it is clear that it represents a client session with a server, in this case, the alarm server. In other cases, the dependency on an external component might be less obvious, such as with `CMSvEntry`, a message server entity, where the internal use of a client-server session by the class is hidden from the client.

The chain of dependencies between the components can quickly complicate how your collective change can be performed. Imagine a situation where a client and a service both de-allocate memory in response to a low memory notification. If the client needs to get the service to perform some actions, such as saving data on its behalf, then there is little point in getting the service to de-allocate memory first. When the client uses the service to save data, the service could well end up re-allocating the memory. In the UI layer, for example, there is little point in notifying a list to redraw itself if a different and overlapping UI component, responding to the same change, forces the list to be redrawn again.

If handled incorrectly, these dependencies can at best reduce performance and, at worst, lead to deadlock, where each component is stuck waiting for other components, or live-lock, where a series of components continuously trigger changes in other components.

One way we could handle these dependencies could be to introduce explicit dependencies between the components. In this situation, you would need to maintain a list of root nodes such that when a change were needed each of these root nodes would be notified. Each root node would then be responsible for notifying its dependent children followed by the children in turn notifying the change to all of their dependents and so on.

Putting such a scheme into place is straightforward to start with and does make it easy to understand the relationships between all the components involved in the change. However this approach has a significant disadvantage of causing the components to be tightly coupled such that it would be very difficult to move or remove a root node from the notifications without adversely affecting its children. On an open operating system such as Symbian OS, with components on a device coming from a wide range of companies and individuals ranging from Symbian and the device manufacturers to network operators and third-party application developers, such a tightly coupled scheme would not be acceptable.

Another alternative would be to have each component regularly poll the value of the state they're interested in. However, as discussed in Chapter 4, this would quickly drain the device's battery. Instead, an

event-driven programming approach is needed that is able to respect the existing dependencies between components by notifying them in a structured manner.

Example

Consider the start up of a device. Instead of the tens of servers and applications started on a real device, we limit ourselves in this example to the following components which start in the following order:

1. The system starter starts the system, and hence the other three components, with the goal of making the device appear to start up as quickly as possible.
2. The Communications Infrastructure creates and initializes all the communication protocols on the device.
3. The phone application allows the user to make phone calls.
4. The task scheduling server schedules tasks to be run at specific times.

For the device we're using as an example here, only half the communication protocols are actually required by the phone application whilst the scheduling server is deemed to be a low priority and hence doesn't need to start running scheduled tasks before the device appears to finish starting up. However, this might well be different on another device so the device start-up architecture needs to be flexible enough to accommodate this.

However, for this specific device one way to optimize device start-up²³ is by changing the device start-up sequence to be as follows:

1. The System starter starts ...
2. The Communications Infrastructure phase 1 which loads just the protocols needed by the phone application and then starts ...
3. The Phone application. The device now appears to have booted but the lower priority functionality should be started in the background so this application calls ...
4. An API introduced into the Communications Infrastructure, `RSocketServ::LoadAllProtocols()` which starts ...
5. The task scheduling server.

This solution has the problem that the responsibility for starting the system is now split between a number of components which adds

²³The user's perception of the phone having been started is when they can start making phone calls using the phone user interface, so to optimize device start-up, the phone user interfaces should be initialized and working as the number one priority with all other components being started afterwards in the background.

complexity to an already complex area of functionality. The supply chain is made more intricate too since the phone application is provided by the device manufacturer whilst the Communications Infrastructure is provided by Symbian. Clearly this doesn't work well for even this specific device let alone for other devices with different start-up sequences.

A second possible solution to performing this optimization would be to use the Mediator pattern [Gamma *et al.*, 1994] with the system starter as the Mediator. If this solution were used then the system starter would be responsible for starting all the components in the correct order as well as calling the new `LoadAllProtocols()` function provided by the Communications Infrastructure to complete its initialization. This would move all knowledge of dependencies during start-up to the system starter, solving some of the problems with the first solution.

However this still requires each component to expose the details of its dependencies to the system starter and for the system starter to be written with an understanding of these dependencies. While this might not initially seem like a big drawback of using the Mediator pattern, it starts to be more problematic if the start-up sequence changes. Due to the tight coupling between the system starter and the other components in the system, when a component changes its dependencies the system starter would need updating. A similar thing occurs when a new component needs to be added to the start-up sequence.

Let's say, for example, a weather forecast application was added as a priority application which, for example, adds new dependencies to the protocols that need to be loaded immediately by the Communications Infrastructure. This would require the system starter to make additional requests to existing and potentially to new components as part of the device start-up for this new component to be initialized ready for immediate use by the end user.

Solution

This solution delegates communication between the components that are interested in a particular state change to a central *coordinator*. This coordinator has the sole responsibility of informing a group of components, the *responders*, in a structured manner that a state change has occurred and collecting their acknowledgements before moving on to the next group of responders. This allows the understanding of what needs to be done for each change to be distributed to the individual components whilst retaining central control over the order in which the notifications are sent out as well as eliminating the need for each component to be aware of the other components interested in the change.

Structure

This results in the structure shown in Figure 6.13, where all the objects involved are located in the same thread. Note the use of *Event Mixin* (see page 93) for all the concrete coordinators and concrete responders to be decoupled.

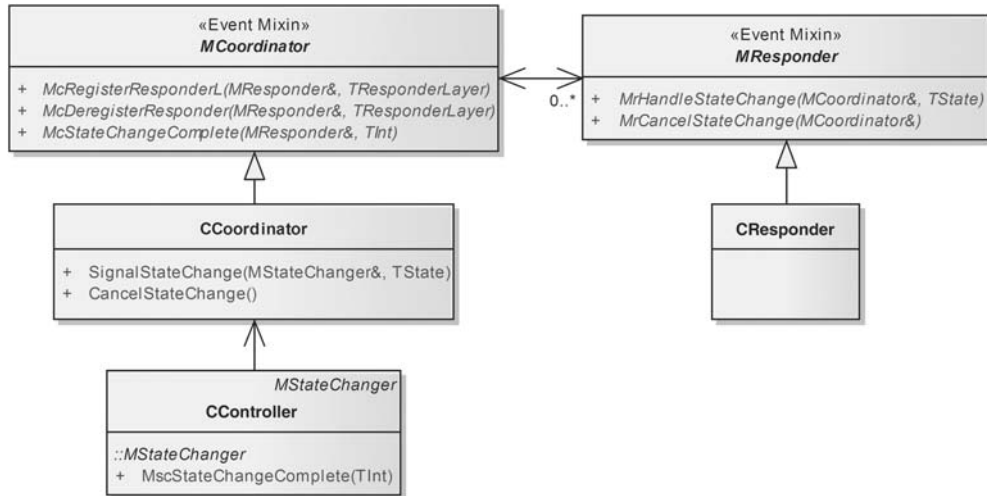


Figure 6.13 Structure of the *Coordinator* pattern (single-threaded)

The controller is responsible for requesting that the coordinator perform a state change across all of the responders. It is important to separate the responsibilities of the controller and coordinator here; the coordinator should be entirely agnostic to what the meaning of the state change is, its role is simply to let all the responders know. All knowledge of when to perform a state change or what to do if one fails should reside in the controller. The controller however should never interact directly with the responders and relies on the coordinator to contact them on its behalf. This separation of responsibilities considerably simplifies the design and allows the system to be loosely coupled.

The responders each register with the coordinator for notification of the state change but when doing so they specify which group, or *layer*, that they wish to be notified as part of. The layers are defined by the coordinator and are usually ordered such that all the responders at the lowest layer are notified first. It is then each responder's responsibility to perform the appropriate actions to ensure that they transition to this new state before acknowledging the change. The coordinator waits for all responders at a particular level to acknowledge the change before

moving on to notify the next layer up and so on. Hence the coordinator's `SignalStateChange()` method is asynchronous since it potentially could take a long time to complete especially if there are a large number of responders.

When using this pattern, it is essential to define two things early on:

- the states you wish to send to the responders
- the layers into which you want to divide up the responders.

These combine to form the *state coordination* model for how the state changes will be performed and coordinated. This is an important decision to make since it is the one thing that all components involved in the pattern will depend on, so changing it later will be problematic. If this model is too abstract then responders will not know how they should respond to each change. On the other hand, if this model is too specific to the first implementation of the pattern then it'll result in tighter coupling than necessary.

For most implementations of this pattern within a single thread there is only one controller and one coordinator. However, this pattern is most useful when responders can be found anywhere in the system, even in other processes. In such cases, an IPC mechanism needs to be introduced to allow this to happen. In order to make it as transparent as possible, Proxy [Gamma *et al.*, 1994] is introduced as shown in Figure 6.14.

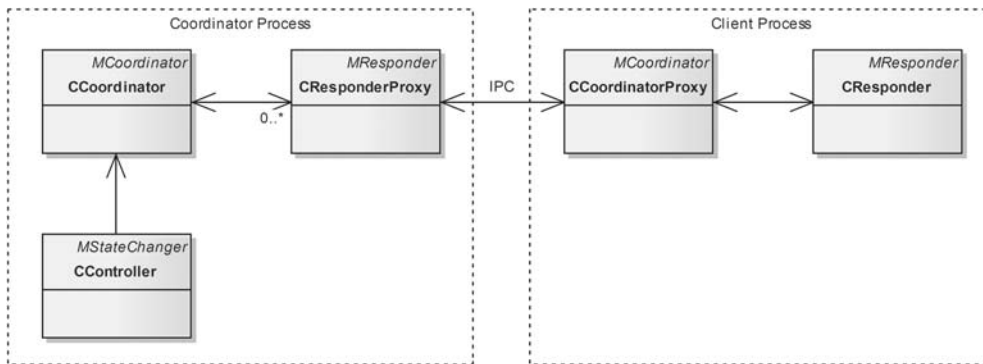


Figure 6.14 Structure of the *Coordinator* pattern (multiple processes)

Responder proxies present the `MResponder` interface to the coordinator but are implemented to forward these calls over an IPC channel to a *coordinator proxy* that makes the corresponding call on its local responder. The calls from the responders follow the reverse path through these objects. Note that there is usually one responder proxy and coordinator proxy per responder since there is normally only one responder per client

process. However, since there are multiple responders in the system, the coordinator is aware of multiple responder proxies.

Dynamics

Figure 6.15 illustrates the sequence of messages in a typical interaction between a controller, a coordinator and three responders operating at two different layers. The proxies are not shown, to avoid obscuring the sequence with unnecessary detail as they simply forward messages over IPC and have no other behavior. No process boundaries are shown although the three responders are likely to be in three different processes.

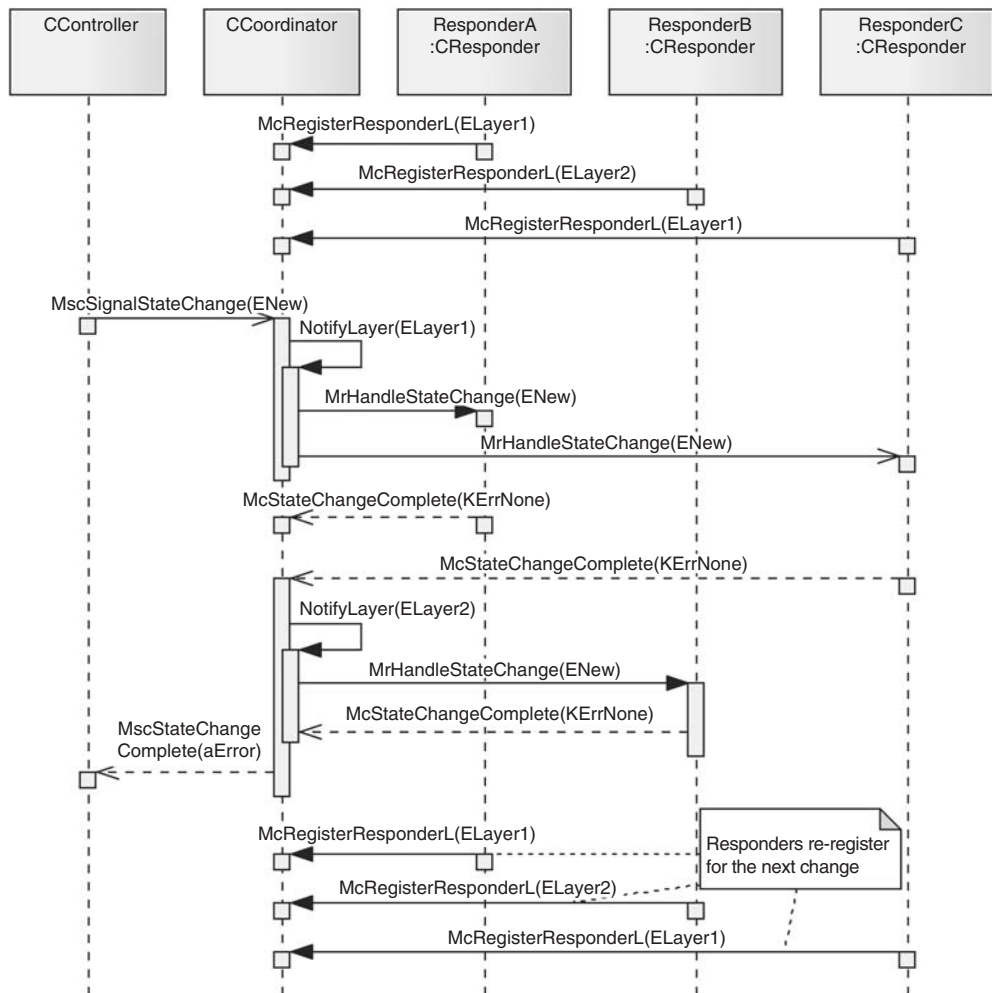


Figure 6.15 Dynamics of the *Coordinator* pattern (successful state change)

The coordinator notifies the registered responders of a change in the order of their registered layers and not in the order that they register. The coordinator waits for the acknowledgement of the notifications before moving on to send notifications to the next layer. Only once the coordinator has received acknowledgements from all the responders does it tell the controller that the state change has completed. Note that the coordinator sends all the notifications for a single layer out at the same time without waiting for each individual responder. This is because we assume that all responders at the same layer are independent of each other and hence we can optimize the notification procedure by parallelizing between the synchronization points between the layers.

Figure 6.16 illustrates how a failure should be dealt with during the state change.

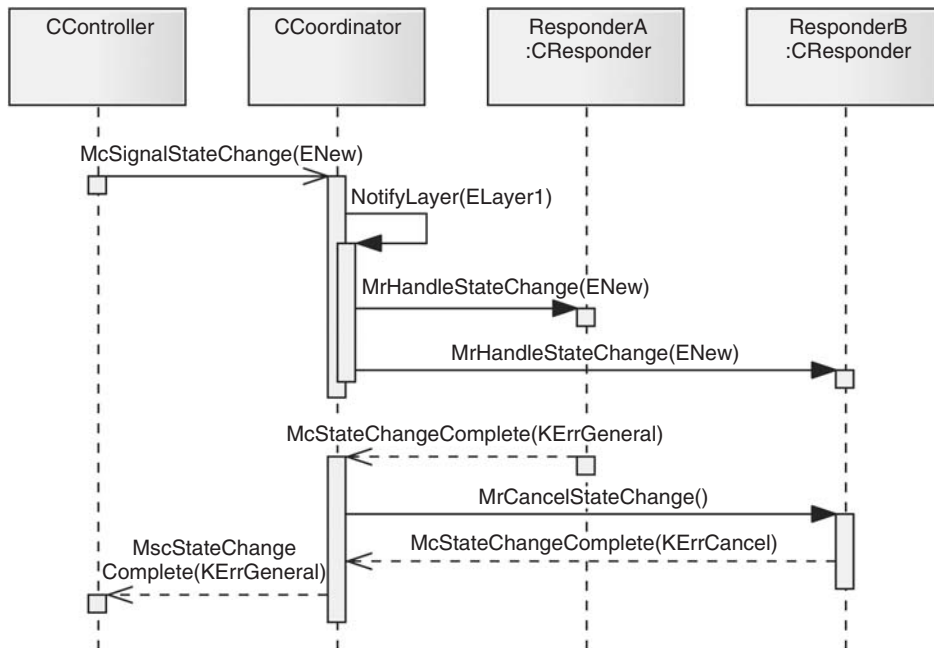


Figure 6.16 Dynamics of the *Coordinator* pattern (failed state change)

When the coordinator receives an error acknowledgement from one of the responders it should cancel any outstanding state change notifications and then call `MscStateChangeComplete(error)` on the controller. This is an *Escalate Errors* (see page 32) approach²⁴ to pass responsibility

²⁴Specifically the ‘Escalating Errors Without Leaving’ variant, since we can’t Leave due to asynchronous calls.

for resolving the error back to the controller so that the coordinator can be completely agnostic to the actual meaning of a state change.

When the controller receives the error it has a number of choices about how to resolve the error. For instance, it could take some corrective action and re-try the same state change or perhaps it could signal a change back to some base state. The exact choice depends very much on what each of the state changes mean.

However, one complication to be aware of is that the responders may not be in a consistent state. Figure 6.16 shows *ResponderA* failing, however, if it had been *ResponderB* that had failed then *ResponderA* would be in a different state to *ResponderB*.

Implementation

Single-Threaded Coordinator

The implementation given here is for the structure described in Figure 6.13.

You first need to decide how to express the state coordination model. The most straightforward way is to use one enumeration for the possible states²⁵ and another for the list of layers. For this example, we choose to model the layering of the responders on the layering of the system as shown below:

```
enum TState
{
    EState1,
    EState2,
    ...
};

enum TResponderLayer
{
    EKernel,
    EBaseServices,
    EAppEngines,
    EAppFramework,
    EAppUI,
    EApplications,
};
```

Once the model has been decided upon, the three main classes can be implemented using the types that describe the coordination model.

First of all the controller, whose main job is to start the state changes, is shown here. This is satisfied within `SignalStateChange()` which simply makes calls to `iCoordinator` as appropriate. As a result of this

²⁵If needed, more complex types can be used to represent each state, so long as they all derive from a single base class.

the controller, is called back via the `MStateChanger` interface which it should implement to handle both the success and failure of the state change:

```
// Event Mixin used by the coordinator to signal completion of a state
// change request
class MStateChanger
{
public:
    void MscStateChangeComplete(TInt aError) = 0;
};

class CController : public MStateChanger
{
public:
    void SignalStateChange();
    ...

public: // From MStateChanger
    void MscStateChangeComplete(TInt aError);

private:
    CCoordinator* iCoordinator;
};
```

Each of the responders derives from the following interface designed using *Event Mixin* (see page 93). Each method takes a reference to an `MCoordinator` object which they later use to send their acknowledgements of the state change. A cancel method is provided since the state change can be asynchronous:

```
class MResponder
{
public:
    // When the state change has been handled, whether or not it was
    // successful, ChangeComplete() must be called on aCoordinator. This
    // can be done synchronously within this function or asynchronously.
    virtual void MrHandleStateChange(MCoordinator& aCoordinator,
                                     TState aNewState) = 0;

    // This will only ever be called when the responder is in the
    // process of handling a state change. It must synchronously call
    // ChangeComplete() on aCoordinator with KErrNone if the state change
    // couldn't be stopped or KErrCancel if it was.
    virtual void MrHandleStateChange(MCoordinator& aCoordinator) = 0;
};
```

A coordinator derives from the following interface designed using *Event Mixin* (see page 93). It allows responders to register and de-register for state changes as well as to signal when their state changes have completed. Note that `MscStateChangeComplete()` is not a leaving function because that would escalate errors from the coordinator to a responder whereas we want most errors in the coordinator to go to the

controller. `McRegisterResponderL()` is an exception to this; if that function cannot be completed for whatever reason, then this is an issue just for that individual responder rather than the whole system.

Note that the `McStateChangeComplete()` function does not have a trailing 'L', which indicates it is not a Leaving function. This is because the function is called by a responder and any Leave would need to be handled by one of them. Instead any error that occurs during the execution of this function needs to be resolved either by the coordinator itself or passed onto the controller.

```
class MCoordinator
{
public:
    void McRegisterResponderL(MResponder& aResponder,
                             TResponderLayer aLayer) = 0;
    void McDeregisterResponder(MResponder& aResponder,
                               TResponderLayer aLayer) = 0;
    void McStateChangeComplete(MResponder& aResponder, TInt aError) = 0;
};
```

The `CCoordinator` class derives from `MCordinator` so that it can work with the responders to change their state. However, its main additional responsibility is to provide an interface for the controller to start (and potentially cancel) a state change. This is done by providing the `SignalStateChange()` method with the `MStateChanger` interface that is used to asynchronously signal the completion of the task.

You might also consider using *Singleton* (see page 346) to ensure that there is only one instance of this class. However, the lifetime of the coordinator would still need to be managed somewhere and since only the controller should directly use the `CCoordinator` class, *Singleton* (see page 346) doesn't benefit us much.

```
class CCoordinator : public CBase, public MCoordinator
{
public:
    static CCoordinator* NewL();
    virtual ~CCoordinator();

    void SignalStateChange(MStateChanger& aController, TState aNewState);
    void CancelStateChange();

public: // From MCoordinator
    void McRegisterResponderL(MResponder& aResponder,
                             TResponderLayer aLayer) = 0;
    void McDeregisterResponder(MResponder& aResponder,
                               TResponderLayer aLayer) = 0;
    void McStateChangeComplete(MResponder& aResponder, TInt aError) = 0;

private:
```



```

if(iController != NULL)
{
    // There's an ongoing transition so complete the request with error
    aController.MscStateChangeComplete(KErrInUse);
    return;
}
// Else start a new state change transistion
iController = &aController;

// Start from the bottom of the dependency layers
iCurrentLayer = EKernel;
iNewState = aNewState;
NotifyCurrentLayer();
}

```

NotifyCurrentLayer() iterates over all the registered responders at the current layer, notifying them all at once and transferring the notified responders to iWaitingResponders to track which responders have acknowledged the state change at the current layer. Note that MResponder::MrHandleStateChange() could be implemented to synchronously acknowledge the state change. Hence we need to check at the end if there are any responders we're waiting for. If not then we immediately advance to the next layer.

```

void CCoordinator::NotifyCurrentLayer()
{
    // Set up the waiting responder array
    iWaitingResponders.Close(); // Free the memory previously used
    iWaitingResponders = iResponders[iCurrentLayer];

    // Force the responders to re-register
    iResponders[iCurrentLayer] = RArray();

    // Notify all waiting responders. This needs to be done in reverse so
    // that any responders which respond synchronously don't interfere
    // with the part of the array being iterated over when they're removed
    // from iWaitingResponders.
    const TInt count = iWaitingResponders.Count();
    for (TInt i = count; i >= 0; --i)
    {
        iWaitingResponders[i]->MrHandleStateChange(&this, iNewState);
    }
    if(iWaitingResponders.Count() == 0)
    {
        // Not waiting for any responses so go to next layer
        NotifyNextLayer();
    }
}

```

NotifyNextLayer() is responsible for notifying the next layer up until the final layer has been done.

```
void CCoordinator::NotifyNextLayer()
{
    ++iCurrentLayer;
    if(iCurrentLayer >= KLastResponderLayer)
    {
        // Finished all notifications so notify the controller
        iController->MscStateChangeComplete(KErrNone);
        iController = NULL;
        return;
    }
    // Else notify the current layer
    NotifyCurrentLayer();
}
```

The next two public `CCoordinator` methods are simple registration and de-registration methods which update the array of responders at the appropriate layer with the change. Note that if a responder wishes to re-register for the next state change it should call this function *before* it calls `MCordinator::MscStateChangeComplete()` for the previous state change. This ensures that there is no opportunity for a notification to be missed by the responder.

```
void CCoordinator::McRegisterResponderL(MResponder& aResponder,
                                         TResponderLayer aLayer)
{
    iResponders[aLayer].AppendL(&aResponder);
}

// Attempting to de-register a responder that isn't registered is a
// programmatical fault so Fail Fast (see page 17)
void CCoordinator::McDeregisterResponder(MResponder& aResponder,
                                         TResponderLayer aLayer)
{
    TInt index = iResponders[aLayer].Find(&aResponder);
    ASSERT(index != KErrNotFound);
    iResponders[aLayer].Remove(index);
}
```

`McStateChangeComplete()` is called by each responder to acknowledge that they've transitioned to the new state they were notified of in `MrHandleStateChange()`. When a responder has responded, it is removed from the list of waiting responders. If this was the last response that the coordinator was waiting for then the next layer is notified of the change.

```
void CCoordinator::McStateChangeComplete(MResponder& aResponder,
                                         TInt aError)
{
    ASSERT(iController); // Check a full state change is in progress
```



```

// Remove the responder from the waiting array. If it can't be found
// then this function has been called incorrectly so panic.
TInt index = iWaitingResponders.Find(aResponder);
ASSERT(index != KErrNotFound);
iWaitingResponders.Remove(index);

if(aError < KErrNone && aError != KErrCancel)
{
    HandleStateChangeError(aError);
    return;
}

// Check to see if waiting for any other responders at this level
if(iWaitingResponders.Count() == 0)
{
    // Not waiting for any more responses so notify the next layer
    NotifyNextLayer();
}
}

```

When an error occurs whilst handling a state change across all the responders, the coordinator has the responsibility of canceling any outstanding individual responder state changes and escalating the error to the controller to resolve properly.

```

void CCoordinator::HandleStateChangeError(TInt aError)
{
    // Cancel all outstanding responder state changes
    const TInt count = iWaitingResponders.Count();
    for (TInt i = count; i >= 0; --i)
    {
        iWaitingResponders[i]->MrHandleStateChange(&this);
    }

    // Check the responders have all responded synchronously to the
    // cancel. They should've each called McStateChangeComplete(KErrCancel)
    // by now, which would've removed them from iWaitingResponders.
    ASSERT(iWaitingResponders.Count() == 0);

    // Finished all notifications so notify the controller
    iController->MscStateChangeComplete(aError);
    iController = NULL;
}

```

Finally, a way is needed for the controller to tell the coordinator to cancel the state change of all the responders:

```

void CCoordinator::CancelStateChange()
{
    ASSERT(iController); // Check a full state change is in progress
    HandleStateChangeError(KErrCancel);
}

```

Multiple Processes Coordinator

To implement this pattern to support responders in different processes requires the addition of responder proxies in the coordinator process and a coordinator proxy in each responder process (see Figure 6.14). Each proxy then implements the necessary IPC to forward a local call to its other corresponding proxy. Symbian OS provides a number of mechanisms which you can use to provide the necessary IPC; however, *Client–Server* (see page 182) is an obvious choice because of the high-level support it offers and the pre-existing client–server framework that helps you to implement it quickly.

Further details can be found in *Client–Server*, however the following points should give some guidelines for applying it to this pattern:

- The coordinator process should host the server with each of the responder proxies acting as a server session object.
- Each responder process acts as a client with a coordinator proxy acting as the client session object.

Using *Client–Server* also makes it possible for you to perform security checks on responders before allowing them to register for state change notifications. If this is needed then you should implement these security checks within the coordinator process since any security checks executed in the client process can be subverted.²⁶

Consequences

Positives

- There is loose coupling between components. By using the coordinator to interface between the controller and the responders there is no longer a dependency between them. This makes it very easy to replace or remove individual responders without needing to change any other components, not even the coordinator or the controller. This also allows additional dependencies to be added easily by just registering another responder at the appropriate layer.
- The testing cost is reduced; a natural consequence of the loose coupling is that it is easy to add test responders.
- The power usage is reduced as a natural result of the extensive use of event-driven programming techniques²⁷ rather than polling for state changes.
- It is easy to change the order in which responders are notified, especially when compared to other potential solutions in which there

²⁶See Chapter 7 for more information.

²⁷See Chapter 4 for more information.

are long chains or trees of components passing on the state change notifications.

- This pattern increases robustness as it has a central error-handling facility within `MStateChanger::MscStateChangeComplete()` which allows for a consistent policy for resolving errors. In addition, the use of this pattern makes the pre-existing dependencies between components visible and hence they are less likely to be accidentally ignored.
- Responsiveness to state changes is increased. The layering specified at run time by the responders allows for as much parallelization of notifications as is possible without breaking the system.

Negatives

- It is difficult to change the coordination model since each component depends on it. Any changes after development are likely to affect all the components involved in this pattern and hence could be potentially quite costly to do if the coordination model is not suitably future proofed.
- Error handling is less flexible. Separating the controller from each of the responders does have the disadvantage of making the error handling in the controller necessarily generic and unable to react to errors with specific responders. This can be partly addressed through the *Status Reporting* extension.
- Rollback of a partially complete state change can be difficult if not impossible. If a responder fails to transition to the new state part way through the whole state change then there will be responders in both the new and the old states, which is unacceptable in most situations. Depending on your situation this could be resolved by re-trying the state change or even asking responders to reset themselves, however there are a number of situations where this will not help.

Example Resolved

Returning to the simplified system start-up example, this pattern can be applied by the system start being the controller. Each application or service involved in the device start up then has the opportunity to be a responder. The coordination model used is as follows:

- There are only two possible states: critical boot and non-critical boot. When a responder receives the critical boot state change it should only do the essential operations required to get the phone part of the device operational. The non-critical boot state change is used to

indicate that components should finish initializing by performing any remaining initialization operations.²⁸

- The Symbian OS system model [Rosenberg and Arshad, 2007] provides a natural layering abstraction. Using the system model for Symbian OS v9.2 as an example, there are five main layers each of which has up to five sub-layers²⁹ within them. This doesn't include applications so we need to add a new top layer to support them. This leads to the following codified coordination model:

```
enum TStartupChanges
{
    ECriticalBoot,
    ENonCriticalBoot,
};

enum TSystemLayers
{
    EKernelServices1,
    EKernelServices2,
    EKernelServices3,
    EBaseServices1,
    EBaseServices2,
    EBaseServices3,
    EOsServices1,
    EOsServices2,
    EOsServices3,
    EApplicationServices1,
    EApplicationServices2,
    EApplicationServices3,
    EUiFramework1,
    EUiFramework2,
    EUiFramework3,
    EApplication1,
    EApplication2,
    EApplication3,
};
```

We still need a coordinator. It would be possible for the system starter also to be the coordinator but Symbian OS already has a server which is suitable for coordinating start-up, called the 'domain manager'. The layers referred to in this pattern are called domains, in the parlance of the domain manager. The relationship between these layers is defined by a domain hierarchy. The domain manager is not provided in standard SDKs so is not available to all Symbian OS developers but its `RDmDomain` (coordinator) and `CDmDomain` (responder interface) classes are available for use by the system starter and responders.

²⁸Responders can use *Episodes* (see page 289) to help them divide the initialization up into separate phases.

²⁹For our purposes, just three sub-layers are sufficient.

The phone application in the example does not need to respond to any of these changes. It simply needs to be fully initialized when started, so it would not be a responder in our example. The remaining components in the example would be responders with layers assigned as follows:

- Communications Infrastructure – `EOsServices2`
- Task scheduling server – `EOsServices3`
- Weather application – `EApplication2`

The ordering of the layers ensures that the communications server will have responded to the `ENonCriticalBoot` change before the weather application is notified, so the protocols required by the weather application will have been loaded by the time it comes to deal with a state change.

Dependencies within the same layer are handled using the sub-layers, represented by the numbers at the end of the names. This ensures that the communications server (at layer 2 in OS Services) will be notified before the task scheduling server (at layer 3 in OS Services).

There is a subtlety here when dealing with the `ECriticalBoot` state change. Since it is the first state, when should it be sent to the system? If it is changed by the system starter before anything is started then there will be no responders registered to receive it, but there is no point in sending this change too late in device start-up.

The domain manager solves this by allowing a responder to request the current state of the domain it is joining. This value would be the last change that was distributed via the domain manager and so allows components that have only just subscribed to obtain the last state change that was performed.

This allows the system starter to request the `ECriticalBoot` change at the beginning of start-up so that when other servers, such as the communications server, are started they can tell from the domain manager that the last change was `ECriticalBoot` and respond accordingly as if they had just received that change.

The code for performing this would be similar to the following, where `CCommsServer` is derived from `CDmDomain` which is derived from `CActive`:³⁰

```
CCommsServer::InitializeStartupCoordinator()
{
    ...
    // First request notification of the next state change
```

³⁰See *Active Objects* (page 133).

```

RDmDomain::RequestTransitionNotification(iStatus);
TInt currentState = GetState();
if(currentState >= ECriticalBoot)
{
    // Already part-way through boot, so perform
    // critical initialization actions
    PerformCriticalInitialization();
}
if(currentState >= ENonCriticalBoot)
{
    // If it is already past ENonCriticalBoot then perform
    // the non-critical initialization.
    PerformNonCriticalInitialization();
}
...
}

```

Care must be taken to request notification of the next change before performing other actions, including acknowledging a notification, to eliminate the possibility of missing any future changes.

This highlights how important it is to define the model correctly. In this case, the definition of the `ENonCriticalBoot` state change means that if a responder first sees `ENonCriticalBoot` then it should perform all the actions for `ECriticalBoot` in addition to the ones for `ENonCriticalBoot`. This is an issue which does not occur in all situations where this pattern is applicable and the relevance of this issue depends on what changes are distributed as well as the relationship between the changes. Here the changes are cumulative whereas in other situations they might not be.

Other Known Uses

In addition to the system start-up example, the *Coordinator* pattern is used in the following examples:

- *View Switching*
The view server acts as a coordinator to allow different application views to be brought to the front of the UI. The coordinator can be accessed through the `CCoeAppUi` class. The application views, derived from `MCoeView`, are each responders, with the views having activation or de-activation state changes requested on them according to which view was requested to be activated by the controller of the view server. Not all registered views are notified on a view change, only the current view and the view which is going to become the current view, so the dependencies model is more complex than given in previous examples in this pattern. Note that the view server uses a timeout when switching views to prevent hung clients from stopping all view-switching activity.

The view server architecture is described in detail in the Symbian Developer Library and the full complexity of it goes beyond what is described here.

- *Save Notifiers*
The save notifications provided by the shutdown server via `CSaveNotifier` is another example of this pattern being used, with 'save data' events being the state changes distributed across the system. This gives components the opportunity to save their data before the device shuts down. The shutdown server acts as coordinator between the responders deriving the `CSaveNotifier` class. The Look and Feel (LaF) shutdown manager generates the save data events and so acts as the controller, with the shutdown server acting as the coordinator.

Variants and Extensions

- *Multiple Controllers*
This pattern assumes that there is only one controller in the system. However, it can be extended such that a controller service is exposed by the coordinator process. This would allow multiple controllers to request state changes from all the responders. One problem with this variation is that it divides up the control responsibility between multiple objects which makes maintenance that much harder. However, this variation can work well when the state changes sent by each controller do not overlap.
- *Response Timeouts*
This pattern is susceptible to being blocked by a responder who doesn't respond to the state change notifications quickly enough or even at all. One way to deal with this is to start a timer once all responders at a particular layer have been notified of the state change. Those that don't respond within a specific time are sent a `StateChangeCancel()` and the entire state change is failed. However, this is a fairly inflexible approach and you should also consider more sophisticated approaches such as first sending an 'Are you alive?' signal to slow responders to give them the opportunity to request more time.
- *Status Reporting*
You may find it necessary for the coordinator to provide a status-reporting service that allows clients, such as the controller, to obtain useful information such as the most recent state change and what its status was. If a state change did fail then you could provide diagnosis information such as which layer it failed in.
This is particularly useful when the coordinator fails to complete a state change across all the responders: they will probably be in

an inconsistent state. With the current pattern implementation when the controller is resolving the error it does not have any information about which responders did manage to successfully complete their state change. This additional service would rectify this.

One danger is that you expose the responders to the controller and hence start introducing dependencies between them and reducing the flexibility of your implementation.

- *Combined Acknowledgement and Re-registration*

This description of the pattern assumes that responders don't always want to re-register for state changes. However, if this is not the case then you may wish provide an additional acknowledgement method to `MCoordinator`. This method would combine the standard responder acknowledgement with a request to re-register the responder for the next state change.

References

- *Client–Server* (see page 182) can be used to implement the IPC needed to use this pattern with responders in different processes from the coordinator.
- *Escalate Errors* (see page 32) is used within this pattern to move the error handling from the coordinator to the controller where it can be handled more appropriately.
- Proxy [Gamma *et al.*, 1994] is used to hide the need for IPC communication.
- Mediator [Gamma *et al.*, 1994] solves a related problem to this pattern. The Mediator object described by that pattern is tightly coupled with the colleagues with which it interacts, with the Mediator typically creating all the colleagues. A Mediator generally performs unique actions on each of the colleagues, understanding that each colleague has a different role in a system. The *Coordinator* pattern is different from the Mediator pattern in that it moves the intelligence and understanding of what to do for particular changes out to the responders, removing almost all understanding from the coordinator. The coordinator in this pattern could be seen as an unintelligent Mediator. An additional difference from the Mediator pattern is the separation of controller from responder; in the Mediator pattern the colleagues are both controllers and responders.

7

Security

This chapter outlines a number of patterns that can be used to secure your components from attack by software agents, known as *malware*. Before we get into the details of the patterns themselves, we cover some of the basics of the platform security provided by Symbian OS v9 so that they are put into context. For a fuller description please see [Shackman, 2006] and [Heath, 2006].

Security on Symbian OS follows a strategy of *defense in depth*¹ in which there exist multiple *security privileges* which each provide the right to perform a specified group of actions or access specific information or resources. These security privileges are known as *capabilities*² on Symbian OS and are assigned to executables during development. During development, you may also assign to an executable a *secure identifier* (SID) or a *vendor identifier* (VID).³ The capabilities, SID, and VID are known collectively as *security credentials*.

In addition to this, Symbian OS platform security defines the following concepts:

- *The smallest zone of trust is a process* – Symbian OS has been designed to control what each process can do. This reflects the fact that a process is the unit of memory protection and this hardware-assisted protection provides the foundation of platform security.
- *Capabilities determine privileges at run time* – whilst both DLLs and EXEs can be assigned security credentials during development, only the security credentials of an EXE translate directly into run-time credentials. This is because the processes created from an EXE are given the security credentials of the EXE. DLLs on the other hand, if loaded by a process, execute with the security credentials of the

¹ [en.wikipedia.org/wiki/Defense_in_depth_\(computing\)](http://en.wikipedia.org/wiki/Defense_in_depth_(computing))

² See [Heath, 2006, Section 2.4].

³ If not assigned explicitly, the SID defaults to the UID3 of the executable whilst the absence of a VID statement means that it does not have a VID.

process. To prevent this allowing a DLL to take on security credentials at run time, the following *DLL loading rule*⁴ is applied: ‘A process can only load a DLL if that DLL has been trusted with at least the capabilities that the process has’.

- *Data caging files* – this allows important files, whatever their content, to be protected. Special top-level directories are provided on each drive that restrict access to the files within them as follows:⁵
 - `/sys` contains executable code and cannot be read from or written to.
 - `/resource` allows only read access.
 - `/private/<own SID>` allows read and write access.
 - `/private/<other SID>` cannot be read from or written to.
 - `/<other>` has no restrictions.

Symbian OS platform security divides components into the following three types:

- *Trusted Computing Base* (TCB) [DoD, 1985] components have unrestricted access to the device resources. They are responsible for maintaining the integrity of the device, and applying the fundamental rules of platform security. The rest of the operating system must trust them to behave correctly and their code has, consequently, been very strictly reviewed. In particular, the TCB guarantees the following:
 - The capabilities, SID, and VID of a process (as returned by the associated `RProcess` and `User` functions) can be trusted by the caller of these functions.
 - The capabilities, SID, and VID of executables, both EXEs and DLLs, cannot be tampered with and then executed by the Loader (part of the TCB). This includes any code loaded via `RLibrary`, `REComSession` or the `RProcess` APIs.
 - Executables on the device cannot be overridden except by legitimate⁶ upgrades for the executable.

For more information on the services that the TCB provides see [Heath, 2006]. Note too that TCB is also used to refer to the capability given

⁴See [Heath, 2006, Section 2.4.5].

⁵Unless the process is highly trusted and has been assigned a specific capability approved by the device manufacturer.

⁶To a limited extent this can be circumvented by using Symbian Open Signed (what used to be known as Developer Certificates). However, this can only be done with the end user’s full knowledge and permission.

to these components, such as the kernel and file server and, on open devices, the software installer.

- *Trusted Computing Environment* (TCE) servers, beyond the TCB, which are granted orthogonal restricted system capabilities. This includes servers providing sockets, telephony, certificate management, database access, and windowing services.
- Everything else.

To build secure software on top of Symbian OS it is particularly important to keep the following principles in mind:

- *Economy of mechanism* [Saltzer and Schroeder, 1975] – this is the practice of minimizing the size of highly trusted code by separating it from code that doesn't require the same level of trust. Such separation enables closer scrutiny and analysis of the security-critical code, increasing confidence that vulnerabilities will not be found after the software has been deployed.
- *Complete mediation* [Saltzer and Schroeder, 1975] – access to a protected resource must always be checked and any temptation to cache decisions for execution performance reasons must be examined skeptically.
- *Open design* [Saltzer and Schroeder, 1975] – security should not depend on keeping mechanisms secret as they are difficult to maintain this way and provide little defense once they've been publicized. You also forego the benefits of others reviewing your design.
- *Least privilege* [Saltzer and Schroeder, 1975] – this requires that each process executes with the least number of capabilities possible so that it only accesses such information and resources as are necessary for its legitimate purpose. This minimizes the damage that can result from security vulnerability in a single component.
- *Privilege leakage* – when a process is trusted with a privilege it is expected to guard the access it is granted by having the privilege. This means that you should carefully consider whether to allow a process to perform an action on behalf of a process that uses a privilege unless it has the same privilege itself.

Having developed a component based on Symbian OS that makes use of specific capabilities you may need to distribute your software for installation on other people's devices. Unfortunately, it's not always possible to simply put it up on a website somewhere and have people download copies to their devices. If your component requires the use of a capability, for whatever reason, then you may be able to rely on the user to grant blanket permission to the application at installation time or 'single-shot' permission at run time. However, if this is not the

case then your component must be *signed*: a process that associates a tamper-proof digital certificate with the component that identifies its origin and allows the software installer to judge whether or not the component should be allowed to install with the security credentials it says it needs.

Executables need to be signed so that stakeholders other than the original developer can judge whether or not they should trust them. For instance, the end user wants to know that their personal data isn't going to be stolen; a content provider wants to prevent unauthorized copies of their music being made, and Symbian and the device manufacturer want to protect their reputation for providing reliable devices. By signing a component with a specific capability, you promise to maintain the integrity of the parts of the device to which the capability gives your component access.

Not all capabilities are equal and the cost of assigning a capability to a component varies considerably. Consider the four main types of capability given below in order of increasing sensitivity:

- *User capabilities* are designed to be meaningful to end users who may be allowed⁷ to grant blanket or single-shot permission for executables to use these capabilities even if they haven't been signed.
- *System capabilities* protect system services, device settings, and some hardware features. Use of these capabilities requires executables to have been signed. The signing process for these capabilities is currently free and can be done locally for testing or for personal use. For commercial use, there are some moderate costs involved and signing cannot be done locally.
- *Restricted system capabilities* protect file system, communications, and multimedia device services. Use of these capabilities also requires code to have been signed. The signing process for these capabilities for any use currently requires a moderate cost and signing cannot be done locally.
- *Device-manufacturer-approved capabilities* are TCB and system capabilities that protect the most sensitive system services. Use of these capabilities requires a device manufacturer to have given permission for the associated application or service to be signed for use on their devices.

Each additional capability that is required may require one or more declarative statements to be filled out or extra testing during the signing process. The more sensitive a capability is, the more likely they are to occur. For further details on how Symbian Signed works see [Morris, 2008].

⁷Depending on the security policy for a particular device.

In summary, components that use the more sensitive capabilities mean additional financial and development costs for the component provider. This is especially important when providing an extension point for which another vendor is intended to supply the plug-ins. For instance, if you were to require plug-ins to have the TCB capability it would mean that most plug-in providers would not be able to target your extension point as they'd not be able to obtain permission for the capabilities required.

Another point to remember about capabilities is that they should only be assigned to a process if they are needed to access a specific API.⁸ Capabilities should not be assigned just to impose additional restrictions on DLLs loaded into the process. For instance, a device manufacturer might be tempted to assign a process the `AllFiles` capability and so prevent third-party DLLs from being loaded. Whilst this would appear to work, in certain circumstances it would be possible to circumvent the restriction if the process doesn't actually call an API which checks `AllFiles` capability. This is because capabilities are not a general measure of trustworthiness. Capabilities were designed to control access to APIs; when a binary is signed with a certain set of capabilities it simply means that it is permitted to access those APIs. Capabilities aren't intended for the purpose of saying 'this code is more trustworthy than that code', they are designed to say 'this code needs to access certain APIs and that code doesn't'.⁹ As an example, if an executable is assigned capability X, which it does not need for any API call, then it could be 'upgraded' to a version that isn't assigned capability X by using Symbian Open Signed.¹⁰ This 'upgraded' version of your component would then execute without any problems but at a lower level of trust which might be enough to allow the malicious developer to be able to install their plug-in into your process and access all its memory.

In most situations, this restriction on being able to assign capabilities isn't a problem because if a process handles sensitive information and so wants to ensure that loaded DLLs can legitimately have access to the data then the process has usually obtained the sensitive data by calling the API of the appropriate service. For instance, if a process handles sensitive device settings then it might have obtained the data originally from the Central Repository which checked it as having the `ReadDeviceData` capability.

When two processes with different levels of trust interact they both need to be wary of what information they send outside of their process, or zone of trust. For instance, Digital Rights Management (DRM)¹¹ data

⁸Whether the process could achieve the same result by calling a different API which doesn't require the capability in question is another issue.

⁹The principle of least privilege.

¹⁰Such signatures are tied to specific devices but for this attack that's more than enough.

¹¹DRM refers to technologies used by publishers and copyright holders to control access to and usage of digital media.

should not be passed outside a process assigned the DRM capability unless the recipient also has the DRM capability. Similarly you should design a process to take care when handling received data. As an example, a DRM process should not only validate all such data, in case it is corrupt, but it should also take into account the level of trust placed by the system in the originator of the data. If a non-DRM process is always allowed to start the playback of DRM material then a malicious application, without even seeing any DRM data, could cause material with a fixed number of uses¹² to become inaccessible.

Now that we've given you an introduction to how security works on Symbian OS, we can move onto the patterns that you can use to build on top of this foundation. *Secure Agent* (see page 240) builds on the concept of processes as zones of trust and describes how to put into practice the 'economy of mechanism' principle when you have to perform one or more actions requiring the most sensitive capabilities.

The remainder of this chapter describes how to support plug-ins to a flexible architecture but without sacrificing security in doing so. Architectural extensibility is a common requirement as it allows your software to be easily upgraded or customized in future. However, by using plug-ins, you introduce new attack surfaces by which malicious software can attack your components. Hence extensibility needs to be balanced with measures to restrict the damage that might be caused by accidental or malicious actions by the plug-ins. In general, the more secure a framework needs to be, the more restrictions it has to place on who can provide a plug-in for it. Without careful consideration of the architecture and design of a solution, it is all too easy to make the restrictions too harsh and unnecessarily limit who can provide plug-ins for your application or service.

The following list summarizes the characteristics of the remaining patterns in this chapter and how they solve this trade-off between flexibility and security:

- *Buckle* (see page 252):
 - There is a low degree of complexity.
 - There are no limitations on interaction between the framework and a plug-in.
 - Plug-ins are not isolated from the framework or each other.
 - The capabilities of a plug-in must match those of the framework.
 - No additional processes are required.

¹²Such material can only be played a finite number of times before the license expires; for instance, you may be able to play a game only three times before you must pay for it.

- *Quarantine (see page 260):*
 - There is a medium degree of complexity.
 - Interaction between the framework and a plug-in is limited to one-off communication at initialization.
 - Plug-ins are isolated from the framework and each other.
 - The capabilities of a plug-in are independent of those of the framework.
 - An additional process is required per plug-in.
- *Cradle (see page 273):*
 - There is a high degree of complexity.
 - There are no limitations on interaction between the framework and a plug-in.
 - Plug-ins are isolated from the framework, and from each other if required.
 - The capabilities of a plug-in are independent of the framework but there are a few restrictions.
 - Additional processes are required.

Whilst reading the following patterns please remember that at no point should any of them be used to breach the security of the device by simply removing an inconvenient security restriction. By doing so you put at risk the security assets of the device, which includes the end user's private data, and may incur costs on the user's behalf. Most of the patterns in this chapter introduce additional process boundaries as part of their solution so that there exist two or more components executing at different levels of trust. In such cases, you need to keep in mind what each process is trusted to do and restrict its functionality to just that and no more. For instance, a framework that handles DRM data and runs plug-ins in a separate sandbox that isn't trusted with the DRM capability should never pass DRM data to the plug-ins. Since there are no technical issues that prevent this, once the framework has got hold of the DRM data, you need to be careful in your designs to prevent this happening. Otherwise the framework would be in breach of the conditions of holding the DRM capability and should not be surprised if a plug-in is created that successfully copies the DRM data that it illicitly handles.

Secure Agent

Intent Minimize security risks by separating out security-critical code into a separate process from non-critical code.

AKA Isolate Highly Trusted Code

Problem

Context

You are developing an application or service which includes functionality requiring high security privileges¹³ together with a similar or greater proportion of functionality which can run with lesser or no security privileges.

Summary

- It is more difficult to find and fix security vulnerabilities in components which include both security-critical and other code ('economy of mechanism').
- Developers of components which require high security privileges should minimize any liability they may be exposed to due to errors in design or implementation ('least privilege').
- You wish to make it as easy as possible to create and distribute updates to your code.
- When using any security privilege, but especially the device-manufacturer-approved capabilities, you are required to ensure that the privileges are not leaked and unauthorized software cannot use them.

Description

The most straightforward architecture for an application or service may be to structure it as a single Symbian OS process or device driver; however if some of its functions require device manufacturer capabilities such as TCB, AllFiles or DRM there are likely to be longer-term drawbacks in having the highly trusted parts of your component executing in the same process context as the less critical parts.

When both highly privileged and non-privileged code are intermingled in the same process context, any analysis looking for security vulnerabilities has to consider all of the code as equally likely to be a

¹³Using APIs which require capabilities granted by the device manufacturer or running in the kernel.

potential source of a vulnerability.¹⁴ When all of the code is running with the same set of capabilities, a vulnerability anywhere in the code could potentially be exploited to use the most sensitive APIs and data accessible by the process. Also, when all the code is packaged for certification and signing as a single component, changes anywhere in the code may lead to the entire component needing to be recertified as requiring the device manufacturer capabilities and not just the highly privileged code. This is true even if the code is separated out into different DLLs since the DLL loading rule would require them to have any device manufacturer capability that the process that loads them does.

Example

The Symbian OS software installer (SWI) is one of the most critical parts of the platform security architecture. It has, in effect, the highest level of security privileges¹⁵ and therefore needs to be trusted at the highest level; it is part of the Trusted Computing Base (TCB) [DoD, 1985] and malicious functionality injected into the TCB can do anything it wants.

At the same time, the software install subsystem covers a lot of functionality that should not be security-critical; it is desirable for clients without device manufacturer capabilities to be able to invoke the software installer and supply a package using various communication and storage mechanisms; it is only necessary that the package being installed correctly passes security checks. The client is also likely to make use of DLLs which are not granted the TCB capability, and thus could not be linked in to a process which has been assigned that capability. This is because almost all the DLLs provided by Symbian OS have all capabilities *except* for TCB, precisely so that they can be widely used but do not have to be audited for use within the TCB.

Solution

Divide your software component into two or more components, with at least one component containing code which is security-critical, and one or more components containing code which is not security-critical. Security-critical code should include both code which requires high security privileges to execute¹⁶ and code which checks the correctness of input data affecting security-related operations.¹⁷ For example, non-critical code could include user-interface components and application logic which only requires access to the application's own private data.

¹⁴Because all the code within a process has access to all the memory used by that process.

¹⁵For instance, it is able to replace or eclipse any executable on the device.

¹⁶For example, accessing protected files.

¹⁷Such as whether an update to a database was generated by an authorized originator.

Dividing your software component into two or more components which execute as separate processes allows each component to be given only the capabilities which it needs to perform the subset of functionality which it implements. By introducing additional processes, each with their own isolated memory spaces, you minimize the damage that can result from a security vulnerability in one component ('principle of least privilege').

Such separation enables closer scrutiny and analysis of the security-critical code, increasing confidence that vulnerabilities will not be found after the software has been deployed. This practice of minimizing the size of highly trusted code is a well-established security principle ('economy of mechanism').

If your component is distributed as an installable package (a SIS file), it will need to be signed by a trusted authority¹⁸ to be granted capabilities. When high security privileges are needed, such as TCB or other device manufacturer capabilities, there are additional steps in the signing process. Packaging the security-critical parts of your component in a separate SIS file allows updates to, and new versions of, the non-critical parts to be created and distributed without requiring the overhead of device manufacturer approval for the new code and hence avoids any additional cost or time so expediting the deployment of urgent fixes.

Structure

Non-critical code and security-critical code are separated into two communicating processes to create two different memory spaces that are isolated from outside interference (see Figure 7.1). This can be achieved with an architecture including two user-mode processes, perhaps using *Client-Server* (see page 182) for the communication between them or, where it is necessary to have code execute in kernel space, implementing the security-critical code as a device driver which communicates with non-critical components running in a user-space process.

The process containing the security-critical components should never simply expose a public API equivalent to an existing API requiring high security privileges to access; if this were the case then any client of that API would have to be just as trusted as the secure agent itself, since any vulnerability in the client could be exploited by an attacker to do the same thing as they could by exploiting a vulnerability in the agent.

Dynamics

Malicious software may attempt to call the secure agent's API as a way of gaining access to protected services or data. The interface between the

¹⁸Typically the device manufacturer or Symbian Signed [Morris, 2008].

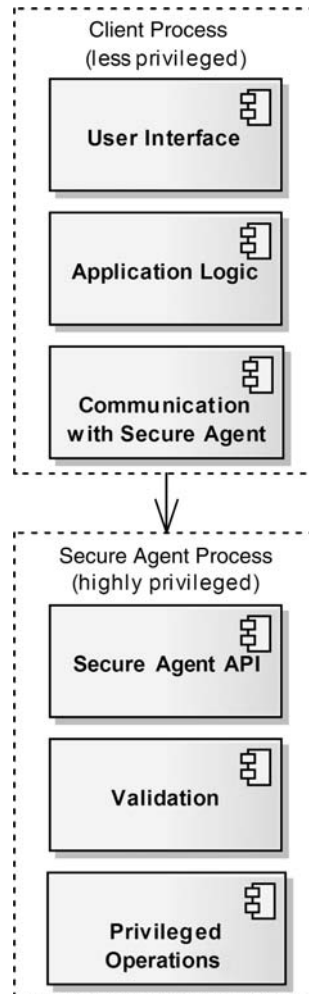


Figure 7.1 Structure of the *Secure Agent* pattern

security-critical and non-critical components must therefore be carefully designed to ensure that communication across the process boundary does not allow unauthorized use of the privileged operations performed by the security-critical components. This undesirable consequence is known as 'privilege leakage', and there are two measures which can be implemented inside the security-critical components to avoid it occurring:

- Ensuring that only authenticated clients are allowed to invoke the security-critical components, for example, by checking that the client has a particular SID from the protected range, mitigates the risk of

malicious software impersonating an authorized client and taking advantage of the APIs exposed by the security-critical components.¹⁹

- Validating the parameters passed to the security-critical components to ensure that they are within expected ranges mitigates the risk of security vulnerabilities caused by unexpected values whether they come from logic errors, user input errors or deliberately exploited security vulnerabilities in an authorized client. Such validation would also include, for example, checking digital signatures on data if the data is expected to be coming from a trusted source.

Figure 7.2 shows the interaction of the non-critical and security-critical processes.

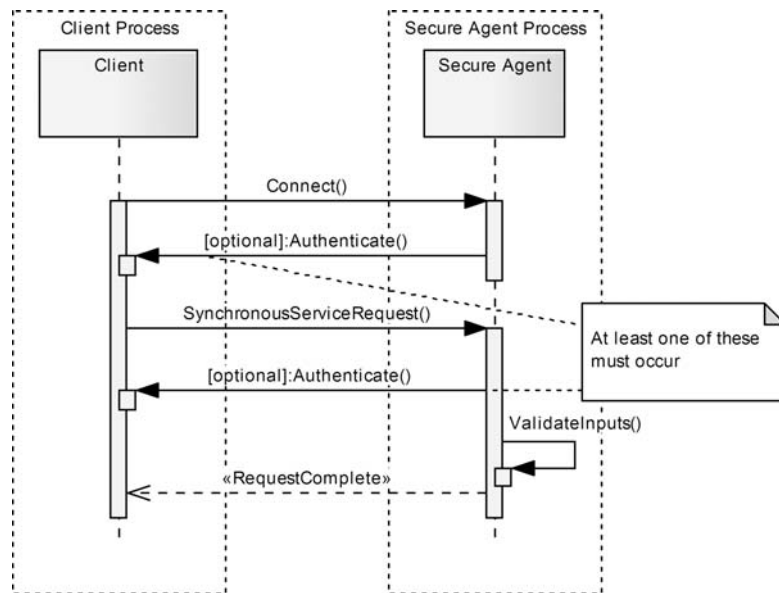


Figure 7.2 Dynamics of the *Secure Agent* pattern

The client process first creates a session with the secure agent which may cause it to be started, if it is not already running. The secure agent can perform a one-off set of checks on the authenticity of the client at this point or can check each service request individually. These checks can include verification of the identity of the client process, via its SID or VID, or that it has been trusted with sufficient capabilities.

When the client invokes the secure agent API to perform actions on its behalf, it can pass parameters with the request that might include, for

¹⁹The `CPolicyServer` class includes a method, `CustomSecurityCheckL()`, which provides a standard way of implementing such checks.

example, basic types, descriptors, and file handles. For each action, the secure agent is responsible for checking the validity of the parameters and any input data read from locations, such as files, provided by the client.

Implementation

Client

The MMP file for the client should look something like this:

```
TARGET      myapp.exe
TARGETTYPE  exe
UID         0 0x200171FD // Protected-range SID

CAPABILITY  NetworkServices, ReadUserData // User capabilities

SOURCEPATH  .
SOURCE      myapp_ui.cpp
SOURCE      myapp_logic.cpp
SOURCE      myapp_ipc.cpp

USERINCLUDE .
SYSTEMINCLUDE \epoc32\include

LIBRARY      euser.lib
```

In this example, the client part of the application requires only user-grantable capabilities which do not require to be signed for. Note, however, that it specifies a protected-range SID which the secure agent can use to authenticate the client, and therefore it needs to be delivered in a signed SIS file.

Secure Agent

The MMP file for the secure agent should look something like this:

```
TARGET      secagent.exe
TARGETTYPE  exe
UID         0 0x200171FE // Protected-range SID

CAPABILITY  AllFiles // Device manufacturer capability

SOURCEPATH  .
SOURCE      secagent_api.cpp
SOURCE      secagent_validation.cpp
SOURCE      secagent_operations.cpp

USERINCLUDE .
SYSTEMINCLUDE \epoc32\include

LIBRARY      euser.lib
LIBRARY      efsrv.lib // Includes APIs requiring AllFiles
```

The secure agent is built as a separate executable. In this example, it requires a device-manufacturer-approved capability, `AllFiles`, so it must either be pre-installed in the device ROM or delivered in a signed SIS file that is approved by the device manufacturer.

Inter-Process Communication

Implementing this pattern requires the introduction of a secure IPC mechanism. Symbian OS provides a number of mechanisms which you can use to provide this, such as *Publish and Subscribe* (see page 114), if you need a relatively simple, event-driven style of communication, or *Client–Server* (see page 182), which supports a more extensive communication style. Which of the various mechanisms you use depends on exactly what needs to be communicated between the client and the secure agent.

Consequences

Positives

- It is easier to concentrate review and testing on the security-critical code, making it more likely that any security vulnerabilities are found before software has been deployed hence reducing your maintenance costs.
- The amount of code running with high security privileges is reduced, making it less likely that there will be undiscovered security vulnerabilities.
- It is easier for signing authorities to assess functionality and grant sensitive capabilities because the amount of code requiring special approval for signing is reduced.
- It is possible to develop and quickly deploy updates to non-critical components without needing to resubmit code for signing approval.

Negatives

- Increased development effort is required at the design stage to divide the functionality into suitable components and to define appropriate APIs for communication between the non-critical and security-critical parts.
- Debugging may be more complex because of the need to trace the flow of control between multiple processes.
- An additional attack surface has been added which can reduce the security benefits.
- The code size of your components will be increased to manage IPC and validate parameters.

- RAM usage will be increased due to the addition of an extra process (a default minimum RAM cost of 21–34 KB²⁰), the increased code size (to implement the IPC channel and the validation of the IPC messages), and the additional objects required in your components and in the kernel to support the IPC.²¹
- Creation of a separate process, marshalling of IPC, parameter validation and additional context switching between the client and the secure agent decreases execution speed.²⁰

Example Resolved

The Symbian OS software installer uses this pattern (see Figure 7.3) to separate out the less-trusted software install functionality, such as reading, decompressing and parsing the SIS file and handling the user interface, to be run as part of the client process. The more-trusted functionality that requires TCB privileges, to write the contents of a SIS file to the file system and update the registry of installed applications, is run as a separate process.

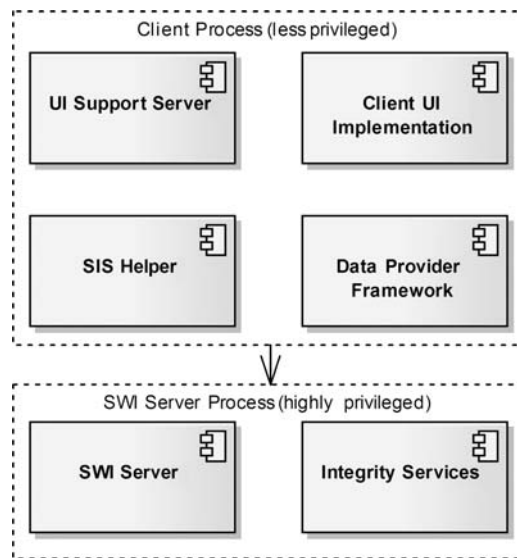


Figure 7.3 Structure of the *Software Installer* pattern

The client process, which only needs the `TrustedUI` capability,²² communicates with the SWI Server using *Client–Server* (see page 182)

²⁰See Appendix A for more details.

²¹For instance, *Client–Server* (see page 182) requires objects in the client, the server and the kernel to model both ends of a client’s session with the server.

²²Available under standard Symbian Signed criteria.

to provide the IPC between the two processes (see Figure 7.4). The main purpose of the SIS Helper component in the client is to facilitate this communication. The SWI Server runs with all capabilities including TCB, DRM and AllFiles as these are needed in order to extract install binaries into the system directories on the mobile device.

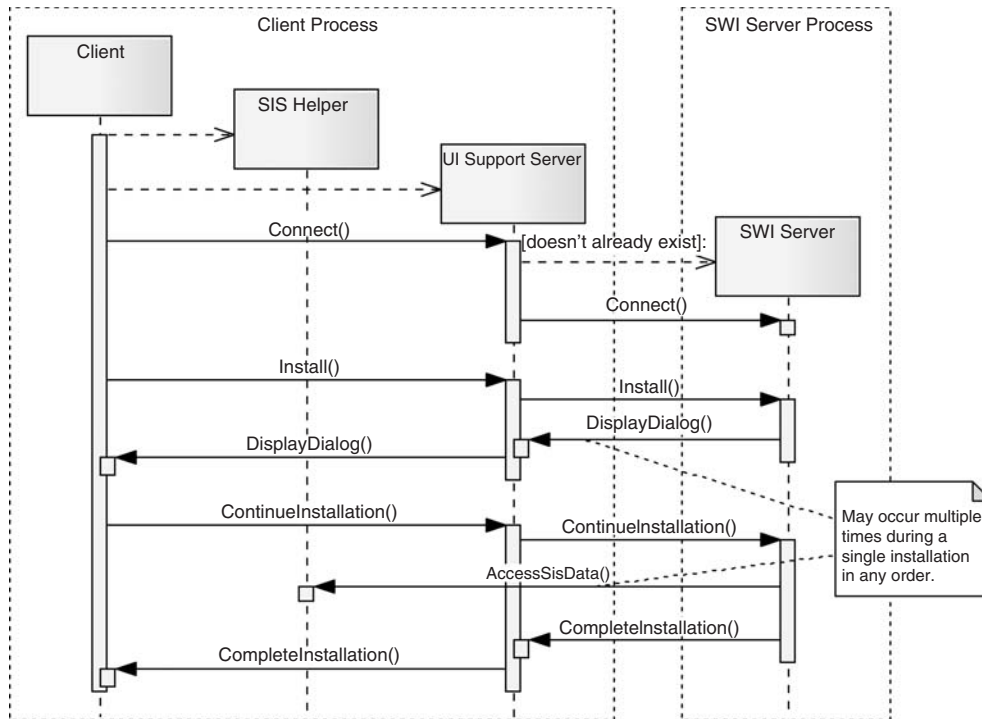


Figure 7.4 Dynamics of the *Software Installer* pattern

Other Known Uses

- *Malware Scanners*

Malware scanners are applications that include an engine to scan for malware as well as components that manage the over-the-air update of malware signature databases and user-interface functionality such as control panels. Only the scanning engine requires high privileges and so it is separated from the other functionality which resides in its own process. The scanning engine however is loaded into the Symbian OS file server process as a PXT plug-in via *Buckle* (see page 252). As the file server is part of the Symbian OS TCB, DLLs it loads, such as these plug-ins, are required to have the TCB capability.

Malware scanner vendors typically package the file server scanning engine plug-in in a SIS file signed with TCB capability, and deliver other components in a separate SIS file that can be updated frequently and easily. So that only a single package needs to be delivered, the scanning engine SIS file can be embedded inside the SIS file for the full package without the need for the outer SIS file package to be signed with device manufacturer capabilities.

Note that the scanning engine is responsible for making sure that any updates done to its database are legitimate before accepting them; end-to-end security of such data updates is typically done by validating a digital signature on the update.

- *Symbian Debug Security Server*

A further example of this pattern that includes highly trusted code running in the kernel is the Symbian Debug Security Server, introduced in Symbian OS v9.4. The low-level operations of the debugger (reading and writing registers and memory belonging to the debugger process, suspending and resuming threads, etc.) are implemented in a Logical Device Driver (LDD) called the Debug Driver, which represents the Secure Agent as described in this pattern. The device driver provides the low-level functions to a process running in user mode called the Debug Security Server. When started, the Debug Driver checks the SID of the client to ensure it is the Debug Security Server so that only this authorized process can access it. The Debug Security Server in turn checks the SID of its clients to ensure that it only provides debug services to authorized debuggers. The Debug Security Server uses a security policy based on access tokens and process capabilities to ensure that a debugger client is only able to access those processes it is authorized for.

Variants and Extensions

- *Separating Security-Enforcing Code from Highly Privileged Code*

In some ways, the desire to group together security-critical code, such as the TCB, to allow in-depth security evaluation is in tension with the desire to isolate code that requires high security privileges to perform its function according to the principle of least privilege. To resolve this tension it may be helpful to consider separating a software component into three subsets (see Figure 7.5).

It is possible to extend the *Secure Agent* pattern to architect an application or service as three communicating processes. The benefits of separately considering the security characteristics of the trusted code (including both security-enforcing and highly privileged code) are maximized by being able to exclude the non-critical code from

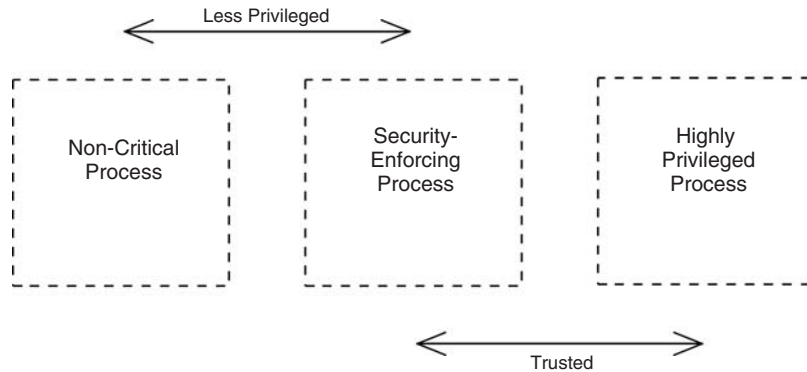


Figure 7.5 Separating security-enforcing code from highly privileged code structure

security review. The benefits of the ‘least privilege’ principle are maximized by separating out the security-enforcing code that does not need high security privileges from the code that must be highly privileged to perform its function.

Factoring the component into three processes, however, magnifies all of the negative consequences listed above so this more complex approach is best reserved for situations where security is of paramount importance. Something analogous to this can be seen in the Symbian OS platform security architecture [Heath, 2006], where the TCB is limited to that functionality which absolutely has to run with the highest privileges and a lot of security-enforcing functionality is provided by system servers in the TCE, which run with only the specific system capabilities they need.

- *Open-Access Secure Agent*
Where the secure agent can safely provide services to any client, the authentication step can be omitted. One example of this could be a service which allows unprivileged clients to initiate playback of DRM-protected content without giving the client any access to the protected data itself. The secure agent still needs to include validation checks on the parameters and other data passed to it to ensure that malicious software cannot exploit any vulnerabilities resulting from processing of out-of-range or other unexpected data. The security implications of such a design should also be carefully considered; if, in our DRM example, the protected content has a restriction on the maximum number of times it can be used (a limited ‘play count’), malicious software could perform a denial-of-service attack by repeatedly requesting playback until all the rights are used up.

References

- *Client–Server* (see page 182) can be used to implement the IPC required between the secure agent and the client processes.
- Protected System [Blakley, Heath *et al.*, 2004] provides some reference monitor or enclave that owns resources and therefore must be bypassed to get access. The *Secure Agent* described here can be seen as an enclave in the Protected System pattern.
- Trusted Proxy [Kienzle *et al.*, 2002] describes a component that acts on behalf of a client to perform specific actions requiring more privileges than the client possesses. It provides a safe interface by constraining access to the protected resources, limiting the operations that can be performed, or limiting the client's view to a subset of the data. The *Secure Agent* described here is an example of this.

Buckle

Intent Load DLL plug-ins, which match your level of trust, into your own process to increase the flexibility of your architecture without compromising security.

AKA None known

Problem

Context

You need to provide a simple-to-use but secure extension point in your component to easily allow future upgrades or customization.

Summary

- Architectural extensibility has to be provided.
- The potential damage caused by plug-ins needs to be limited by minimizing the capabilities with which they execute (principle of least privilege).

Description

A plug-in is a component loaded into a framework at run time so as to provide a certain, usually very specific, function. Frameworks support plug-ins for many reasons which include:

- enabling yourself or other vendors to provide additional functionality that extends an application or service
- easing porting of the application or service to different devices.

One possible way to achieve this is by having a configuration file that allows new settings to be specified after a component has been released. However, this is normally quite restrictive and so a mechanism by which additional code can be plugged in is needed.

This problem is seen by all types of developer whether they're a device creator or a third-party application developer.

Example

If you are creating a service for a mobile phone, then it's unlikely that your code will include a full graphical UI. Despite this, there are occasions

where you still need to be able to display notifications and prompts to the end user. For instance, when the Messaging subsystem detects that an SMS has been received, it needs to be able to alert the user with a ‘Message received’ notification.

To help fulfill this requirement, Symbian OS provides a Notification Server that exposes the `RNotifier` API. However, whilst this works for a number of use cases, it doesn’t support:

- UI-specific dialogs that reflect the look and feel of a device as specified by the manufacturer or operator
- application-specific dialogs provided by third-party developers.

Clearly, to support the above, the Notification Server needs to be extensible. However, to allow untrusted code to provide dialog plug-ins would mean that a malicious attacker could deliberately attack the Notification Server by creating a high-priority notifier which monopolizes the screen. This would prevent any lower-priority notifiers from being activated and potentially make the device unusable: the extension point needs to be protected from being misused.

Solution

Use the ECom service²³ to provide the general extension point functionality of selecting and loading plug-ins as DLLs that expose the plug-in interface specified by your framework. This forces plug-ins to have the same capabilities as the framework itself.

The name of the pattern reflects this purpose: buckles are about joining two objects together (pieces from two different buckles can’t be connected together) and they’re used in seat belts to keep people secure.

Note that this is the most frequently used of all the secure plug-in patterns.

Structure

This pattern has the simple structure shown in Figure 7.6.

Once each plug-in is loaded, it resides within the framework process and can be used exactly as if it had been statically linked to during development. However this structure has significant security implications:

- A plug-in must satisfy the DLL loading rule or it cannot be loaded into the framework process. This rule states that the plug-in DLL must have the same, or a superset of, the capabilities of the loading process.

²³See [Harrison and Shackman, 2007, Section 19.3] for details about ECom.

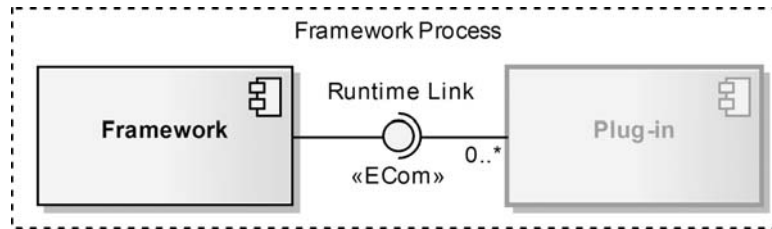


Figure 7.6 Structure of the *Buckle* pattern

- You have chosen to welcome the plug-in into your process and hence do not get any of the benefits of memory isolation offered by executing in a different process. It means that a plug-in has access to any memory used by the framework since Symbian OS only enforces memory isolation *between* processes. It also means that once loaded no further security checks can be made by the framework on a plug-in.²⁴
- Calls by a plug-in to other processes can be authenticated as normal. However, these checks are made on the credentials of the framework process and not anything specified by the plug-in provider.
- It is not possible to authenticate a plug-in by an SID or VID as these are properties of a process at run time and not of code in a DLL.

As far as security is concerned everything hinges on the capabilities you, as the framework provider, assign to the framework process. When you assign these capabilities you should follow these guidelines:

- Assign the minimum set of capabilities needed (principle of least privilege).
- Assign the capabilities required by the framework to meet the security policies imposed by the APIs it calls.
- Assign the capabilities required by a plug-in to meet the security policies imposed by the APIs it calls.
- Do not assign a capability solely to impose an additional restriction on who can provide a plug-in to your framework because whilst this may appear to work there are situations in which it can be subverted.

In the advent of someone trying to spoof the framework this structure does not prevent a plug-in from being loaded. Assuming the plug-in can be loaded then it'll be run with the capabilities that the loading process has been able to obtain. The platform security architecture of Symbian OS

²⁴The plug-in can access and change any stack memory, including return values from a function checking a security policy.

allows us to assume that the loading process has obtained its capabilities legitimately. Hence it can only run the plug-in successfully if it has been trusted with the capabilities used by the plug-in when calling APIs and no advantage is gained by the spoof framework.

Dynamics

It is when a plug-in is loaded that the security check on the plug-in occurs. A plug-in is only successfully loaded if the plug-in's DLL has at least each of the capabilities assigned to the framework EXE. Once loaded, a plug-in is within the process' isolation boundary and will potentially have access to all memory and resources of the framework process. No further security checks can be performed on it after this point.

Plug-in selection is performed via the ECom service which resides in the ECom server process (see Figure 7.7) and hence involves a context switch when the plug-in is loaded. However, once the ECom plug-in has been selected it acts just as if the DLL providing the plug-in was loaded directly into the process and has the same performance benefits as statically loaded DLLs.

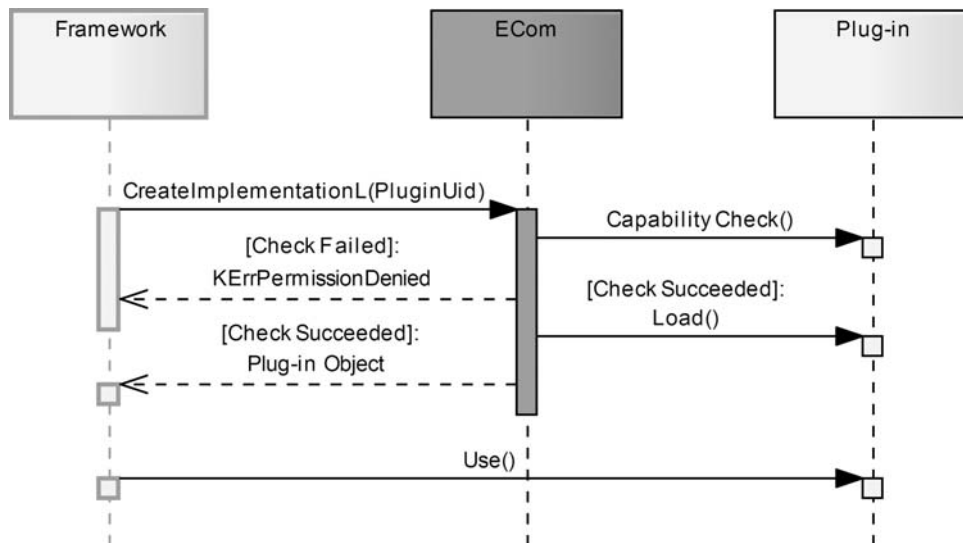


Figure 7.7 Dynamics of the *Buckle* pattern

The framework can use the `REcomSession::ListImplementationsL()` API to find plug-ins that have implemented the plug-in interface specified by the framework provider. For usability reasons, it only returns references to plug-ins that would satisfy the DLL loading rule for the framework process. Whilst this might seem to make the `CreateImplementationsL()` security check unnecessary, it avoids a

Time-of-Check-Time-of-Use (TOCTOU)²⁵ security hole since a client can pass any plug-in UID into `CreateImplementationsL()`.

Implementation

Framework

One of the early decisions that framework providers need to make when using this pattern is what capabilities they should require plug-ins to have.²⁶ Note that this decision has important consequences for who can provide plug-ins and the cost associated with doing so. See the introduction to this chapter for more details.

Once the decision has been made on what the capabilities of the framework will be, the developer of the framework should assign them to the EXE:

```
// Framework.mmp
TARGET      framework.exe
TARGETTYPE  exe
UID         0xE8000077

CAPABILITY    <capability list>

SOURCEPATH  .
SOURCE      framework.cpp

USERINCLUDE .
SYSTEMINCLUDE \epoc32\include
SYSTEMINCLUDE \epoc32\include\ecom

LIBRARY      euser.lib
LIBRARY      ecom.lib
```

For more details of the MMP file syntax, see the Symbian Developer Library.

The loading of the actual plug-ins should be done using the ECom service as described in the Symbian Developer Library and [Heath, 2006, Chapter 6]. A plug-in can be considered to be a resource and hence the patterns in Chapter 3 may be helpful.

Plug-ins

A plug-in developer should assign the same capabilities as used by the framework process, unless there's a very good reason to add more,²⁷ to

²⁵en.wikipedia.org/wiki/TOCTOU

²⁶The complete list is provided in [Heath, 2006, Section 2.4] and the Symbian Developer Library.

²⁷For instance, the plug-in DLL may contain code that is known to be loaded elsewhere. Even in this case you should consider separating the code out into two DLLs, one containing the plug-in and the other containing the code loaded elsewhere.

the plug-in's MMP file to be successfully loaded by the framework at run time.

Other than this, a plug-in should be implemented exactly as specified by the ECom service.

Consequences

Positives

- This pattern is simple to understand, which reduces maintenance costs.
- It is easy to implement the security checks in the framework since it could be as little as one extra line in an MMP file compared to not having any checks at all.
- There is no impact on performance or memory usage except for a small overhead when the plug-in is loaded compared to statically linking to it.

Negatives

- The security check is all or nothing.
- Once a plug-in has been loaded, it can access any data that the framework process can access and hence is able to affect its behavior.
- If the framework process requires additional capabilities to perform its own tasks then this may result in unnecessary security requirements being placed on the plug-ins.
- Adding more capabilities to the framework executable is, at best, difficult. This is because all plug-ins would also need to have this new capability or the framework will no longer be able to load them. The more plug-in providers there are, the more of a problem this is. In practice, adding more capabilities after a framework has been released isn't feasible. If you think you might need to do this then use *Quarantine* (see page 260) or *Cradle* (see page 273).
- Any plug-ins with more capabilities than the framework process cannot use them since a plug-in DLL is limited at run time to just the capabilities assigned to the framework process.

Example Resolved

The Notification Server resolves its need for secure plug-ins by using the ECom service to identify and load plug-ins directly into the `eiksrvs.exe` process. The server requires the TrustedUI capability to allow it to display

dialogs to the user that won't be misleading or somehow corrupt the UI. In addition, the process requires the ProtServ capability to allow it to register the Notification Server within a protected name space and so limit the scope for the server to be spoofed.

The result of this is that dialog plug-in providers must create an ECom plug-in DLL which implements a custom notifier class by deriving from `MEikSrvNotifierBase2`. For the DLL to successfully load into the `eiksrvs.exe` process, it must have the same capabilities as the process (TrustedUI and ProtServ) or more. Since both of these are system capabilities, some additional development time is required of plug-in providers to achieve this but no additional run-time costs are incurred.²⁸ Note that we would expect plug-ins to need the TrustedUI capability since it is directly related to what a plug-in does. The ProtServ capability however, isn't needed by the plug-in though it must have it assigned so that the framework can trust the plug-in not to subvert the framework's own use of the capability.

The result is that third-party developers can add new dialogs to the Notification Server but, because they are signed, any such plug-ins that are found to be malicious could potentially have their signatures revoked in future to prevent them from being installed on new devices.

For more on the Symbian OS Notification Services, including notifier code examples, see [Willee, Dec 2007].

Other Known Uses

This pattern is widely used but we've just picked the following two examples to further illustrate it:

- *Y-Browser [Silvennoinen, 2007]*
This is a file browser application designed for S60 3rd edition devices. In particular, it supports plug-ins for its 'Open With' menu option that are loaded using the pattern. The framework process is created when `YuccaBrowser.exe` is executed, which uses ECom to locate any plug-ins that have at least the following capabilities: NetworkServices, LocalServices, ReadUserData, WriteUserData, UserEnvironment. Since these are all user-grantable capabilities, it is relatively easy for any third-party developer to obtain them and so extend the functionality of Y-Browser without risking the loss of the user's information.
- *Protocol Modules*
Another example is the Symbian OS Communication Infrastructure, which loads protocol module plug-ins (.PRTs) into the `c32exe.exe` process. This framework process has an extensive list of capabilities of up to but not including device-manufacturer-approved grantable

²⁸This is true, at the time of writing, when you use the Open Signed – Online option.

capabilities which means that the providers of plug-ins won't generally include the average third-party developer. Note that, like a number of older Symbian OS plug-in frameworks, the Communication Infrastructure doesn't use ECom but uses `RLibrary::Load()` directly.

Variants and Extensions

None known.

References

- *Client–Thread Service* (see page 171) solves a different problem but its implementation has some similarities to this pattern.
- *Quarantine* (see page 260) is an alternative way of providing an extension point which doesn't require the plug-in to be signed with the same capabilities as the framework when you require little or no communication between the framework and its plug-ins.
- *Cradle* (see page 273) extends *Quarantine* (see page 260) to support an ongoing communication session between the framework and its plug-ins.
- See Chapter 3 for patterns describing how you can improve execution performance or RAM usage by changing the point at which you load and unload plug-ins.
- The following documents contain some additional information on ECom:
 - Symbian Developer Library » Symbian OS guide » System Libraries » Using ECom
 - Symbian Developer Library » Examples » ECom example code

Quarantine

Intent Load plug-ins as separate processes, operating at different levels of trust to your own, to increase the flexibility of your architecture without compromising security.

AKA None known

Problem

Context

You need to provide a secure fire-and-forget extension point in your component and you can't use *Buckle* (see page 252) due to the restraints it places on the capabilities of the framework and its plug-ins.

Summary

- Architectural extensibility has to be provided.
- The potential damage caused by plug-ins needs to be limited by minimizing the capabilities with which they execute (principle of least privilege).
- There must be minimal restrictions on who can provide plug-ins.
- Different sets of capabilities are required for plug-ins and the framework.

Description

Buckle (see page 252) works best when there is a close match between the capabilities of the framework and those expected of its plug-ins. Where there is a mismatch between the capabilities of the framework and any that one or more plug-ins might need at run time, it can be tempting to simply raise the capabilities of the framework process so as to support any reasonable activity of a plug-in DLL. This is bad practice since the result is to raise the security risk by unnecessarily putting more trust in the whole framework process and all the DLLs it depends on,²⁹ not just the plug-ins. It would also be contrary to the principle of least privilege. For instance, if one plug-in needs TCB then creating a useful framework would be significantly more difficult if the plug-in was loaded as a DLL.

Alternatively, the framework may need a hard-to-obtain capability for its own use which is unrelated to the trust required of the plug-ins and

²⁹As a direct result of the DLL loading rule.

you don't want to restrict who can provide plug-ins by forcing them to be signed with a capability that their code doesn't need to operate correctly.

Another consideration is that the scope for security problems is significant when the plug-ins reside in the same process as the framework as in *Buckle* (see page 252). This is because the smallest unit of memory isolation is the process and so you might have invited a wolf in sheep's clothing into the fold!

Example

Most devices have a control panel through which various system settings can be configured. For instance it might allow you to change the system time and date or set the Bluetooth device name. Moreover, the set of things that can be configured vary from device to device. If there isn't any support for Bluetooth then there's no need to change the Bluetooth device name, etc. Symbian OS v8, before the advent of platform security, provided support for control panels through the classes `CApaSystemControlList` and `CApaSystemControl` which can be found in `epoc32\include\apgctl.h`. In particular, these classes allowed the UI layer to find and load at run time³⁰ plug-in DLLs that provided individual control panel items, such as the Bluetooth and Time dialogs. Anyone could provide their own control panel plug-in and so affect the device configuration. Such a plug-in could easily be of poor quality and adversely affect the device by accidentally deleting all network access profiles, for instance.

When platform security was introduced in Symbian OS v9, continuing to allow plug-ins to be provided as DLLs – effectively using *Buckle* (see page 252) – was not acceptable. The key to understanding this is to consider what capabilities you'd give to the process loading the control panel plug-ins. `ReadDeviceData` and `WriteDeviceData` are obvious choices but `LocalServices` is also needed to change the Bluetooth setting. Is there a GPS chip in the device? If so, the `Location` capability might be needed. The problem is that the Control Panel has such a diverse set of possible needs that its process would soon end up needing virtually all capabilities. This would not just present a valuable target for malicious software to try to subvert but would also force providers of simple control panel dialogs to get their plug-ins signed with a number of restricted system capabilities. They might not be able to justify the expense of obtaining them when their software doesn't require the capabilities to perform its functionality.

Instead another way is needed to allow plug-ins to be loaded whilst breaking the dependency of a plug-in's capabilities on those provided by the framework.

³⁰Via a mechanism conceptually similar to ECom though much simpler.

Solution

The solution presented here allows plug-ins to operate at a different level of trust from the framework. This is achieved by implementing plug-ins as separate processes that are created on demand by the framework process which decouples the framework and the plug-ins to allow them to have completely different protected memory spaces and capabilities.

This is reflected in the name of the pattern because, between the framework and the plug-ins, there is a barrier that prevents infection, known as malicious attacks, from spreading in either direction.

Structure

The framework creates plug-ins at run time by calling one of the `RProcess::Create()` APIs resulting in Figure 7.8.

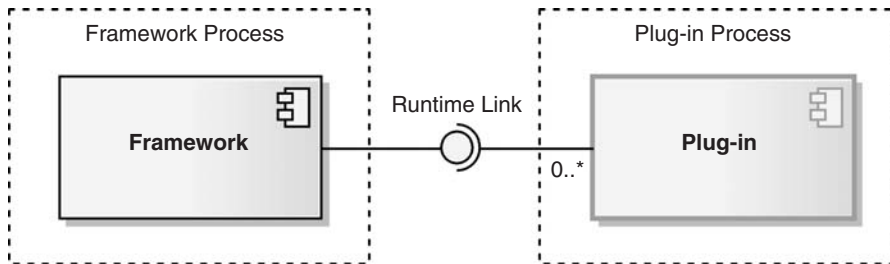


Figure 7.8 Structure of the *Quarantine* pattern

Note that it is not enough for a plug-in to be in a separate thread. This is because a thread containing a plug-in in the same process as the framework would still be able to access the memory of the framework thread as well as having the same capabilities as the framework process. The execution time overhead of crossing a process boundary is always the same as or greater than that of simply crossing a thread boundary. For most current devices, the difference between the two is not significant, compared to the total time taken,³¹ so this shouldn't be a problem. In addition, the use of this pattern assumes that there isn't any extensive communication between the framework and a plug-in anyway.

Interestingly, this pattern can be seen as the inverse of the *Secure Agent* (see page 240) in that we are separating out the less trusted components into their own processes.

³¹This is true for the multiple-memory model used on ARMv6 processors which are currently the most common CPUs on Symbian OS devices. Unfortunately this isn't true for older devices using the moving-memory model on ARMv5 processors where the context switch between processes is 35 times slower than between threads in the same process (see Appendix A for more details).

Dynamics

Figure 7.9 shows how a plug-in process can be loaded. It uses the `RProcess::Rendezvous()` synchronization technique to co-ordinate the initialization of a plug-in process. This works in the same way as thread synchronization does and is described in more detail in the Symbian Developer Library.

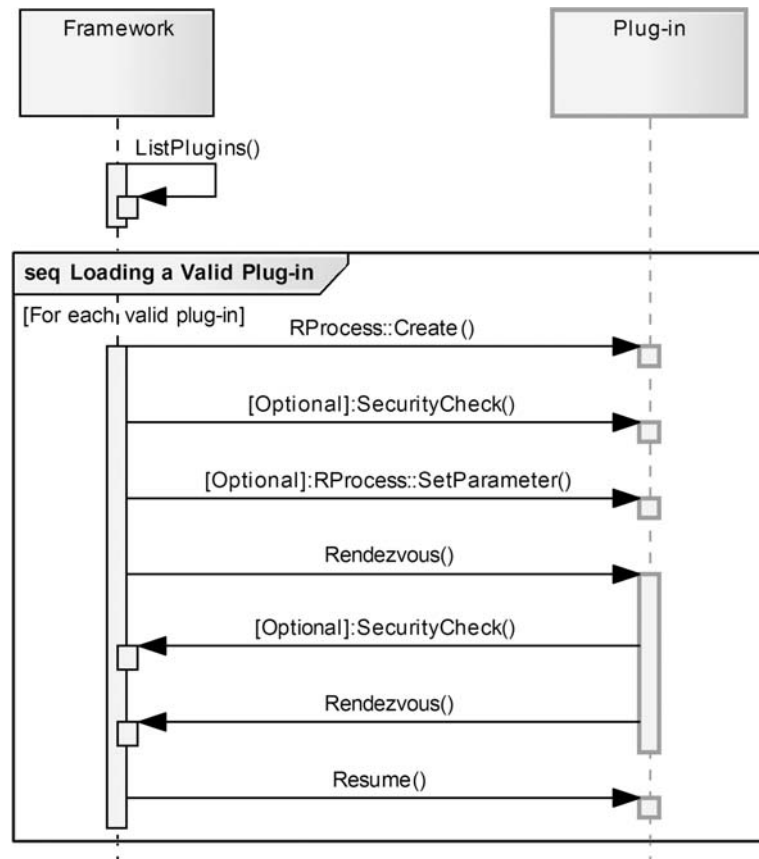


Figure 7.9 Dynamics of the *Quarantine* pattern

As you can see from the diagram, both the framework and a plug-in have the opportunity to authenticate each other by performing security checks during the plug-in loading procedure. These security checks are an essential part of mitigating the security risk that a process other than the framework may attempt to load a plug-in. To avoid this, a plug-in process must check that it hasn't been invoked and passed information by a process spoofing the framework. If the security checks fail, the plug-in process should end its initialization procedure prematurely. The

framework however has an advantage, in that it is able to perform its security check on a plug-in process before the plug-in has a chance to execute any code. This allows the framework to `Kill()` a plug-in if it fails the security checks.

This solution only allows a limited amount of communication between the framework and each plug-in once a plug-in has been created. This is because any communication channels needed have to be set up during the loading of each plug-in. Since you only need a fire-and-forget extension point, as assumed in the context, this shouldn't be an issue. If you do need to maintain significant communication between the framework and a plug-in then you should consider using *Cradle* (see page 273) instead of this pattern.

Implementation

Framework

Before a plug-in can be loaded, you need to be able to list all the available plug-ins that implement the interface required by the framework. Since we're not loading DLLs, ECom can't be used – you need to provide your own alternative mechanism. One way would be to specify that when a plug-in is installed a resource file should be placed in a standard location with a specific name, such as `\resources\<framework name>\<framework UID3>\plugin<plug-in UID3>.rsc`. Note that the UID3 values are used to ensure that the directory and each individual plug-in RSC file are unique. A plug-in's resource file needs to contain at least the name of the EXE file that can be used to create the associated plug-in process. You can then use the `TFindFile` class to search for it.

The following is an illustration of how to implement the loading of a plug-in within `framework.exe`:

```
_LIT(KPluginCommand, "<static plug-in initialization data>");

// This function is passed in the plug-in to load which has been
// identified elsewhere
void CFramework::LoadPluginL(const TDesC& aPluginName)
{
    RProcess plugin;
    User::LeaveIfError(plugin.Create(aPluginName, KPluginCommand));
    CleanupClosePushL(plugin);

    // (1) Security checks on the plug-in should be performed here using
    // the RProcess functions HasCapability(), SecureId() and VendorId().
    // For example, the following code checks that the plug-in
    // has the ReadUserData capability:
    if (!plugin.HasCapability(EcapabilityReadUserData,
        __PLATSEC_DIAGNOSTIC_STRING(
```



```

        "Checked by CFramework::LoadPluginL"))
    {
        User::Leave(KErrPermissionDenied);
    }

    // (2) Additional arguments can be passed to the plug-in process at
    // this point using the RProcess::SetParameter() functions. For
    // example, you could pass over some dynamic data:
    TPckg<TDynamicPluginData> pckg(TDynamicPluginData(...));
    plugin.SetParameter(KProcessSlotDynamicPluginData, pckg);

    TRequestStatus stat;
    plugin.Rendezvous(stat); // Synchronize with the plug-in process
    if (stat != KRequestPending)
    {
        plugin.Kill(0); // Abort loading the plug-in
        User::Leave(stat.Int());
    }
    else
    {
        plugin.Resume(); // Logon OK - start the plug-in
    }
    User::WaitForRequest(stat); // Wait for start or death

    // We can't use the "exit reason" if the plug-in panicked as this
    // is the panic "reason" and may be "0", which cannot be
    // distinguished from KErrNone
    TInt ret = (plugin.ExitType() == EExitPanic) ? KErrGeneral : stat.Int();
    User::LeaveIfError(ret);

    CleanupStack::PopAndDestroy(); // plug-in
}

```

The security checks should be performed as specified at (1) in the above code. The TCB is used to perform the security checks on your behalf, which guarantees that they cannot be tampered with. For instance, if the executable containing the plug-code is on removable media, the TCB checks that it hasn't been tampered with off the device.

Note that the above allows for limited communication between the framework and a plug-in. At (2) the framework can pass information to a plug-in process via 16 possible environment slots.³² These can be set by the framework to contain items such as file handles; handles to other system resources (such as global memory chunks, mutexes and semaphores); integers, descriptors, and TRequestStatus objects, amongst others. In return, a plug-in process can use these same mechanisms to pass information back to the framework so long as the framework is expecting that this might happen. One example of this would be:

1. The framework passes a file handle to a plug-in on creation and calls `Rendezvous()` to wait for the plug-in process to initialize itself.

³²Fewer than 16 are available if the plug-in is also an application since the application framework uses some for its own purposes. See `CApaCommandLine::EnvironmentSlotForPublicUse()`.

2. During the plug-in's initialization, the plug-in writes data using the file handle. When it is finished, it calls `Rendezvous()`.
3. When the framework process resumes, it assumes the plug-in has finished with the file handle and reads the new data that it points to.
4. The data provided by the plug-in is validated before being used.

For more information see the Symbian Developer Library.

Plug-ins

When creating a plug-in, you first need to decide what capabilities to assign it. This will depend entirely on what functionality you need it to perform and the cost of signing your plug-in. Once you've chosen the capabilities, you need to assign them to your plug-in code via a `CAPABILITY` statement in the MMP file for your plug-in EXE. For more details of the MMP file syntax, see the Symbian Developer Library.

The next step is to implement the process entry point as illustrated by the following code snippet:

```
Static void RunPluginL()
{
    // (3) Security checks on the framework should be performed at this
    // point using the User functions CreatorHasCapability(),
    // CreatorSecureId() and CreatorVendorId(). For example, the
    // following code checks that the framework has the LocalServices
    // capability:
    if (!User::CreatorHasCapability(ECapabilityLocalServices,
                                   __PLATSEC_DIAGNOSTIC_STRING(
                                       "Checked by RunPluginL")))
    {
        User::Leave(KErrPermissionDenied);
    }

    // (4) Any arguments passed to the plug-in from the framework can
    // be accessed here using the User functions: CommandLineLength(),
    // CommandLine, ParameterLength() and the Get...Parameter() variants.
    // For example:
    TBuf<512> cmd;
    User::CommandLine(cmd);
    // Parse contents of cmd

    // Get additional arguments from the environment slots
    TPkgBuf<TDynamicPluginData> dataPkg;
    User::LeaveIfError(User::GetDesParameter(KProcessSlotDynamicPluginData,
                                             dataPkg));

    TDynamicPluginData& data = dataPkg();

    // Initialization complete, now signal the framework
    // (5) Implement here any plug-in-specific functionality that the
    // framework expects to happen synchronously.
    RProcess::Rendezvous(KErrNone);
}
```

```
// (6) Implement here any plug-in-specific functionality that the
// framework expects to happen after the plug-in has finished loading.
// As much of the implementation should be executed here as possible to
// minimize the period during which the framework is blocked waiting for
// the Rendezvous() above.
}

// Entry point for the plug-in process
TInt E32Main()
{
    __UHEAP_MARK;

    CTrapCleanup* cleanup=CTrapCleanup::New();
    TInt ret=KErrNoMemory;
    if (cleanup)
    {
        TRAP(ret, RunPluginL());
        delete cleanup;
    }

    __UHEAP_MARKEND;
    return ret;
}
```

The security checks should be performed at (3) in the above code before any further action is taken since if the creating process isn't trusted you shouldn't even look at the data passed across. Even if the creating process does pass the authentication checks, you should validate any input data before using it; this not only makes your plug-in more secure but also more reliable.

Consequences

Positives

- This pattern allows the plug-ins to run under the capabilities of their own choosing, independent of what the framework specifies. This leads to both the framework and each plug-in running with the minimum capabilities that they individually, and not collectively, need. The main benefits of this are that:
 - the security risk to the device is reduced
 - it is easier for developers to provide a plug-in.
- There is a reduced security risk to the framework since the plug-ins run in a separate memory space.
- There is a reduced security risk to the plug-ins since they each run in a separate memory space from the framework and the other plug-ins.
- The capabilities of the framework can be increased without impacting the plug-ins, which makes maintenance easier.

Negatives

- An additional attack surface has been added, which increases the security risk although this is mitigated by the lack of any ongoing communication between the framework and the plug-ins.
- It only directly supports very simple, one-shot communication between the processes via parameters and return values during the loading of the plug-ins.
- Memory usage is increased due to the addition of an extra process (a default minimum RAM cost of 21–34 KB³³), though this may just be a transitory increase if plug-ins are short lived.
- The responsiveness of the framework is decreased since the creation of a whole new process for each plug-in means that loading a plug-in takes approximately 10 milliseconds longer.³³ Whether this is significant depends on how frequently a plug-in is loaded.

Example Resolved

When the Control Panel framework was upgraded in Symbian OS v9 to reflect the increased security concerns, it was decided to use this pattern to provide the solution. Since control panel plug-ins are responsible for interacting with the user, rather than just changing the plug-ins to be separate processes they were changed to be separate applications. In Symbian OS v9, this meant they'd be separate processes too.

In this example, the framework process is the TechView³⁴ UI layer process `shell.exe` which provides the main screen after the device has booted. This runs with the `PowerMgmt`, `ReadDeviceData`, `WriteDeviceData`, `DiskAdmin`, `SwEvent`, `ReadUserData` and `WriteUserData` capabilities because it provides the main GUI to the user. This process uses an updated version of the `CApaSystemControlList` class to identify applications that have registered themselves as providing control panel functionality by marking themselves with `TApaAppcapability::EControlPanelItem`:

```
void CApaSystemControlList::UpdateL()
{
    RApaLsSession  appArcSession;
    User::LeaveIfError(appArcSession.Connect());
    CleanupClosePushL(appArcSession);

    User::LeaveIfError(appArcSession.GetFilteredApps(
        TApaAppcapability::EControlPanelItem,
        TApaAppcapability::EControlPanelItem));
}
```

³³See Appendix A for more details.

³⁴The Symbian reference device UI.

```

TApAppInfo aInfo;
// Fetch the control panel information one by one and add a
// corresponding control to the control list
while(appArcSession.GetNextApp(aInfo) == KErrNone)
{
    // Update list
}
}

```

The above code means that when a new control panel application is installed a new icon appears in the Control Panel, as shown in Figure 7.10.



Figure 7.10 Control-panel application icons in a) S60 and b) UIQ

Applications that have been identified as providing Control Panel functionality are modeled in the framework process by an updated version of the `CApaSystemControl` class. When a Control Panel icon is clicked, the following code is executed:

```

void CApaSystemControl::CreateL()
{
    RApaLsSession appArcSession;
    User::LeaveIfError(appArcSession.Connect());
    CleanupClosePushL(appArcSession);

    TThreadId threadId;
    CApaCommandLine* commandLine = CApaCommandLine::NewLC();
    commandLine->SetExecutableNameL(iFullPath);
}

```

```

commandLine->SetCommandL(EApaCommandRunWithoutViews);
User::LeaveIfError(appArcSession.StartApp(*commandLine, threadId));
CleanupStack::PopAndDestroy(2, &appArcSession);

// Log on to the newly started control panel thread and wait
// till it exits
RThread thread;
User::LeaveIfError(thread.Open(threadId, EOwnerThread));
TRequestStatus status;
thread.Logon(status);
User::WaitForRequest(status);
thread.Close();
}

```

The above code results in a separate application being created that is displayed at the forefront of the screen. The framework synchronously waits until the user has dismissed it before continuing and hence has no need to `Rendezvous()` with a plug-in process as described in the pattern solution above.

Note that no security checks are applied to a plug-in process. The framework relies on the platform security architecture to prevent the plug-in from doing anything it doesn't have the capabilities for. The worst thing that can happen is that the user is confused by a plug-in's dialog boxes in some way and won't run it again. Since these things only happen when the user clicks on a particular control panel icon why would any 'malicious' code bother? The Control Panel doesn't need to trust its plug-ins; they just have to be trusted by the operating system itself.

One of the Control Panel plug-ins is the Bluetooth settings dialog which has responsibility for gathering Bluetooth configuration settings from the end user as well as calling the relevant APIs to pass on this information to the Bluetooth subsystem (see Figure 7.11).

This Bluetooth control panel plug-in application is run with the `NetworkServices`, `LocalServices`, `WriteDeviceData` and `NetworkControl` capabilities. Note that this means it could not be loaded as a DLL into the framework process since it doesn't have the `PowerMgmt` capability. Even if it did have sufficient capabilities to be loaded as a DLL, it wouldn't be able to perform its functionality at run time because the framework process doesn't have the `LocalServices` capability. Hence both the framework and this plug-in are trusted to perform actions the other is not trusted to do.

This plug-in chooses not to authenticate the process starting it since executing the process does not affect any security asset, such as whether Bluetooth is enabled or not, without the end user giving confirmation by clicking the Change or Edit button shown in Figure 7.11. Security is still maintained because only the end user can affect the security assets through the plug-in. The worst that could happen if the plug-in was used by a malicious application is that it could be repeatedly started and so

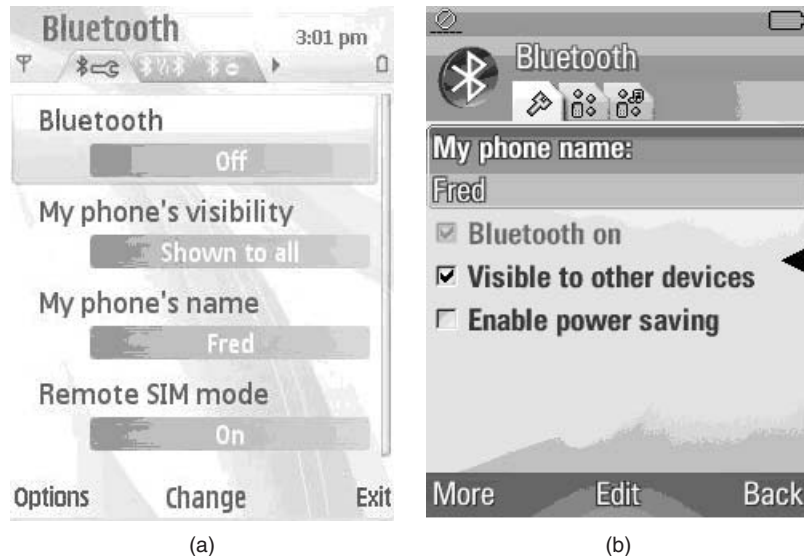


Figure 7.11 Bluetooth settings dialog on a) S60 and b) UIQ

annoy the end user because the screen would keep showing the Bluetooth configuration dialog. This problem isn't specific to Control Panel plug-ins and could happen to any application. The solution under platform security, in both cases, is for Symbian Signed to prevent such malicious code from being signed and to revoke any signed applications that are subsequently found to act in such a manner. Unsigned applications can be installed on Symbian OS devices but the end user is warned that what they're doing could be unsafe.

Other Known Uses

- *Messaging Initialization*

Another use of the pattern can be found in the Symbian messaging architecture. A default message store³⁵ is created if the message server finds either no message store or a corrupt message store when it starts. As the final step, the messaging server then checks whether an executable `mailinit.exe` is present on the device. If it is present, the server starts this plug-in executable to allow customization of the mail store. The behavior of `mailinit.exe` is defined by the UI family of the device. The typical behavior is to perform specific initializations for each message type. For example, the SMS plug-in typically creates a default service entry.

From a security viewpoint, the interesting thing about `mailinit.exe` is that the messaging server doesn't have to worry about whether

³⁵With just a root entry and any standard folders defined in the resource file `msgs.rsc`.

or not it can trust it, because no information or requests are passed to the plug-in. Any customization of the messaging store performed by `mailinit.exe` has to be done by the plug-in process itself opening a session with the messaging server, and whether its actions are valid is determined solely by its own capabilities, not by anything that its parent does.

Variants and Extensions

- *Combined Buckle and Quarantine*
In this variant, a framework chooses to load plug-ins either via *Buckle* (see page 252) or via *Quarantine* (see page 260). This means that the framework loads plug-ins as DLLs into the framework process when a plug-in's capabilities are compatible with those of the framework. Plug-ins are only provided as EXEs when the capabilities between them and the framework process are incompatible. This means the overhead of creating a new process for a plug-in is avoided unless absolutely necessary.

References

- *Buckle* (see page 252) is an alternative way of providing an extension point that is a simpler but less flexible solution to this problem.
- *Cradle* (see page 273) extends this pattern to support an ongoing communication session between the framework and its plug-ins.
- Chapter 3 contains patterns describing how you can improve execution performance or RAM usage by changing the point at which you load and unload plug-ins.
- See the following document for more information on resource files: Symbian Developer Library » Symbian OS guide » System Libraries » Using BAFL » Resource Files.

Cradle

Intent Host DLL plug-ins in separate processes, operating at different levels of trust to your own, to securely increase the flexibility of your architecture whilst maintaining communication with each plug-in.

AKA None known

Problem

Context

You need to provide a secure extension point in your component that supports extensive communication between the framework and the plug-ins whilst not overly restricting the plug-in providers.

Summary

- Architectural extensibility has to be provided.
- The potential damage caused by plug-ins needs to be limited by minimizing the capabilities they execute with (principle of least privilege).
- It's desirable that the restrictions on who can provide plug-ins are as light as possible.
- You wish to allow plug-ins to operate with a different set of capabilities than the framework.
- You need to support a full communication channel between the framework and the plug-ins after the plug-ins have been loaded.

Description

If you're reading this pattern then you've probably tried to get *Buckle* (see page 252) to work but found that it doesn't provide a solution to this problem since you need to allow the plug-ins to operate at a different level of trust to the framework.

Quarantine (see page 260) is an alternative that does allow plug-ins to operate at a different level of trust to the framework but it only directly supports very simple, one-shot communication between the framework and each plug-in; in this context, we need to support a full communication channel between them. Another potential problem with *Quarantine* (see page 260) is that it works best when plug-ins are used infrequently

because the overhead of creating a new process for each use will become prohibitive.

Example

Examples of this problem often come from components that have used *Buckle* (see page 252) but have later found that the constraints it imposes are unacceptable. This can be because the framework doesn't provide sufficient capabilities to satisfy a new plug-in or that the time and cost of signing sufficiently trusted plug-ins is found to be too high.

One such example was the SyncML subsystem which implements a platform-independent information synchronization standard [OMA, 2002]. In particular, it contains a process called the Sync Agent which synchronizes the data on a device with the data on a remote server. A data provider interface is used to support the synchronization of data sources, such as the contacts and calendar databases, on the device. Since such data sources vary from device to device or could even be installed at run time, the interface is required to be implemented by plug-ins.

In an early version of the SyncML subsystem, data provider plug-ins were implemented as ECom plug-ins and loaded into the Sync Agent process as per *Buckle* (see page 252). However, as this process used more than just the user-grantable capabilities this made it difficult for third parties to provide plug-ins and so an alternative design was needed in which plug-ins aren't unnecessarily forced to have at least the same capabilities as the framework. *Quarantine* (see page 260) could have been used to resolve the capability-mismatch issue but that approach doesn't satisfy the need for frequent communication between the framework and its plug-ins carrying potentially large amounts of data.

Solution

This solution allows plug-ins to execute at a different level of trust to the framework but to still maintain ongoing communication between the two. This is achieved by executing plug-ins within separate processes from the framework which therefore execute in separate memory spaces from each other. What makes this pattern different from *Quarantine* (see page 260) is that instead of the framework process using the `RProcess::Create()` API to pass information once to a plug-in, it makes calls on a local proxy that forwards messages over IPC to a plug-in. Similarly, when a plug-in wishes to respond it makes calls on its local proxy which then forwards the response over IPC to the framework.

This is reflected in the name of the pattern because the plug-ins are wrapped in cotton wool and as much as possible is done for them in their *cradle*. The analogy though doesn't fully convey the security risk plug-ins could pose.

Note that this secure plug-in pattern is used only occasionally.

Structure

The main components involved in this pattern (see Figure 7.12) are:

- *Plug-in Client*, which uses the functionality provided by the plug-ins
- *Framework proxy*, which forwards the local function calls made on it by a framework proxy over IPC to the cradle process; there is one framework proxy per plug-in
- *Plug-in proxy*, which forwards the local function calls made by a plug-in to it over IPC to the framework process; there is one plug-in proxy per plug-in
- *Plug-ins*, which are hosted in one of the cradle processes so that they are separated from the framework whilst still providing the configurable functionality needed by the framework.

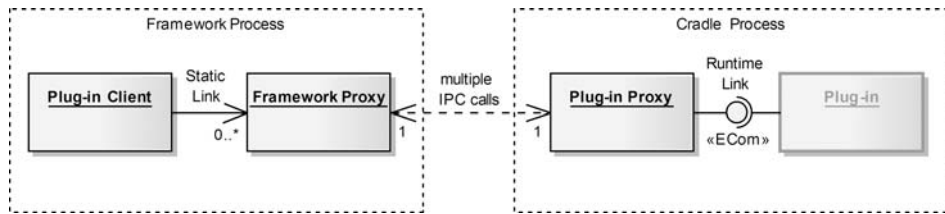


Figure 7.12 Structure of the *Cradle* pattern

All components are supplied by the framework provider with the exception of the plug-ins, of course. This is so that these components only have to be written once and can then be re-used for each of the plug-ins.

Note that the two proxies in Figure 7.12 make it appear to their respective clients as if making calls on them is exactly like making calls on the eventual recipient of those calls in the other process. This is achieved by each proxy implementing the interface of the intended target and simply forwarding the messages on exactly as in the Proxy pattern [Gamma *et al.*, 1994]. The main advantage of this is that the clients of the proxies do not need to have any knowledge of where the recipient resides nor any knowledge of the exact mechanism for communicating over IPC. This simplifies the implementation and allows more flexibility in the architecture.

The exact packaging of this pattern depends on the capabilities you expect the plug-ins to use. The problem is that capabilities are set at compile time, not run time, so when creating the extension point, the framework provider needs to decide which cradle EXEs with which capabilities to provide before any plug-ins exist:

- If you expect all plug-ins to have the same capabilities then only a single cradle EXE with this set of capabilities needs to be provided.

- If you expect the plug-ins to have a wide variety of capabilities, then you will need to carefully choose which cradle EXEs with which capabilities to provide. Ideally, a cradle EXE would be provided for each set of capabilities required by a plug-in but this could quickly become unmanageable³⁶ and you may need to compromise and provide a reduced set of cradle EXEs. The trade-off is that some plug-ins would have to be signed with capabilities they don't need simply because there isn't a cradle EXE that exactly matches their required capability set which may mean additional time and cost for plug-in providers. However, a good choice of cradle EXEs that keeps in mind the different types of capabilities (user, system, restricted system, device-manufacturer-approved) should minimize this impact.

Having decided which cradle EXEs need to be supplied, the framework provider needs to decide whether or not to host multiple plug-ins in a single cradle process. Irrespective of how many cradle EXEs you have, it is possible to host each plug-in in its own process simply by using the appropriate cradle EXE to create another instance of a process when each new plug-in is loaded. From a security point of view, this is a preferred option since each plug-in is completely isolated from interference from other plug-ins. The problem with that approach is, if you need to have multiple plug-ins loaded at once, this might have a significant RAM overhead³⁷ since each process will consume a default minimum of at least 21–34 KB.³⁸ If this is a significant problem then hosting multiple plug-ins in a cradle process is the only way to reduce this.

The rest of this pattern assumes that there could be multiple cradle EXEs each used to create a process hosting several plug-ins. This then covers the other possibilities as degenerate cases.

Interestingly, this pattern can be seen as the inverse of the *Secure Agent* (see page 240) in that we are separating out the less trusted components into their own processes.

Dynamics

Figure 7.13 shows the two main sequences – loading a plug-in and making use of it. In Figure 7.13, `RProcess::Create()` is shorthand for the full process start-up sequence described in *Quarantine* (see page 260). An actual plug-in is loaded by the cradle process using *Buckle* (see page 252).

³⁶Since there are currently 20 different capabilities this means there are 2^{20} distinct capability sets possible.

³⁷There is also the execution time overhead of creating a process though this is less likely to be significant and you may be able to use a pattern from Chapter 3 to address the problem without compromising security.

³⁸See Appendix A for more details.

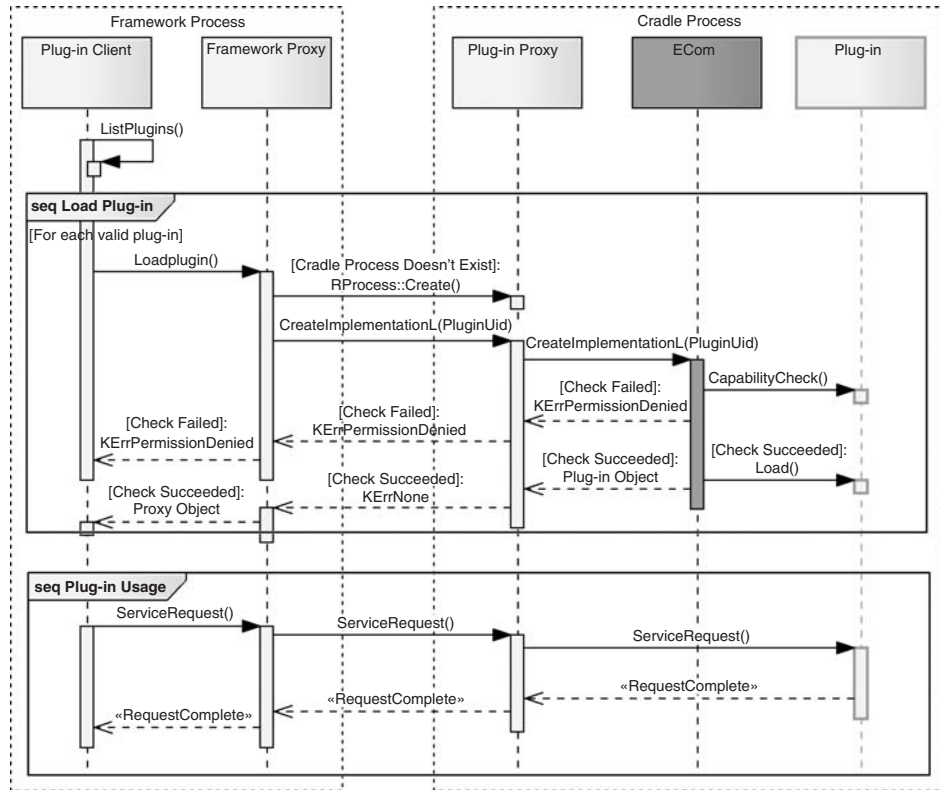


Figure 7.13 Dynamics of the *Cradle* pattern

When loading a plug-in, the plug-in client ensures itself that it is loading a sufficiently trusted plug-in via the following authentication chain:

1. The plug-in client can trust the framework proxy because they reside in the same process and hence must both have been assigned the capabilities of their process.
2. The framework proxy has the opportunity when creating a plug-in process, and hence the plug-in proxy, to authenticate it using its capabilities, SID, or VID just as is done in *Quarantine* (see page 260).
3. The plug-in proxy relies on the DLL loading rule to prevent it from loading plug-ins that do not have at least the capabilities of the plug-in process.

A plug-in can be sure it has been loaded by a trusted framework due to the following authentication chain:

1. Any untrusted process would not be able to execute with the necessary capabilities needed by a plug-in to perform its activities.

2. The plug-in process, hence the plug-in proxy, has the opportunity to authenticate its creator just as is done in *Quarantine* (see page 260).
3. The framework proxy can trust the plug-in client because they reside in the same process and hence must both have been assigned the capabilities of their process.

Implementation

Framework

The first thing to do is to determine which capabilities might be required by the plug-ins. This will then allow you to choose the number and the capabilities of the cradle EXEs needed. Note that you should be cautious in providing cradle EXEs at different levels of trust (i.e. with different capabilities) to avoid over-complicating the implementation. In addition, this architecture is easily extended with additional cradle processes in future if needed.

If more than one cradle EXE is required for your framework, then you need to choose a mechanism by which plug-in providers can specify which of the cradle processes their plug-in should be loaded into. One way this can be achieved is by requiring plug-in providers to supply two different files:

1. A resource file that tells the framework which cradle process should be used to load a plug-in. This can be done in the same way that *Quarantine* (see page 260) uses resource files to identify plug-ins.
2. A standard ECom resource file that tells a cradle process how to load a plug-in using ECom.

The reason that two resources are needed is that if the framework process were to call `REComSession::ListImplementationsL()` to discover what plug-ins are available on the device then the list returned would be limited to just those that could be loaded into the framework process. Since the framework and cradle processes have different capabilities, the plug-in list returned might not include plug-ins that could be legitimately loaded by the cradle process in question.

Once you have identified which plug-in to load and decided when to do so, you should load it as follows:

1. If the required cradle process does not already exist it should be started in the same way plug-in processes are started in *Quarantine* (see page 260) so that each process has the opportunity to perform security checks on the other.
2. Over IPC, request that the cradle process loads a plug-in by specifying its ECom implementation UID.

3. The cradle process should load a plug-in via ECom as described in *Buckle* (see page 252).

Symbian OS provides a number of mechanisms which you can use to provide the necessary IPC:

- Message Queues (described in the Symbian Developer Library)
- *Publish and Subscribe* (see page 114)
- *Client–Server* (see page 182)

Of these, the preferred solution is *Client–Server* because Message Queues doesn't provide as much support for exceptional conditions. For instance, if a plug-in panics its process then, with Message Queues, the framework would only know about this because it wouldn't get any responses to its service requests whereas *Client–Server* ensures that the framework is told as soon as a client ends a session with it for any reason. Since you're hosting plug-ins in a separate process from the framework, you probably suspect that this is an increased possibility. *Publish and Subscribe* could be made to work but, since it is not intended for extensive two-way communication, it is not immediately amenable to addressing your needs here.

The details of the implementation of IPC via *Client–Server* are explained in detail on page 182 although the following points give you some guidelines for adapting it for this pattern:

- The framework proxy implements the client-side interface by deriving from `RSessionBase`.
- The server side should be implemented to execute within each of the cradle processes. Assuming there's no use for the plug-ins once the framework is no longer using them; the server should be implemented in a transient³⁹ manner so that the cradle process is only running when there are plug-ins loaded. Making the server transient helps to reduce the RAM used by this solution at the cost of increased time to load a plug-in.
- The cradle server should only allow clients with the framework's SID to connect to it. This not only enforces this design constraint but also simplifies the implementation since there's unlikely to be a need for further security checks on individual IPC calls. Hence using the `CPolicyServer` class is probably not going to be necessary.
- If you are hosting multiple plug-ins in a single cradle process then it would make sense to implement the plug-in proxies as sub-sessions rather than sessions.

³⁹For more details, see the transient server variant of *Client–Server* (page 182).

One drawback of using *Client–Server* (see page 182) is that to protect the server from being spoofed you need each cradle process to be assigned the system capability `ProtServ` which then forces each plug-in to be assigned it too. This grants the right to the cradle server to register within a protected name space. If `ProtServ` weren't used then a piece of malware could register a server with the same name as the cradle server which would cause the cradle process to fail to start. Hence this spoofing would result in a denial-of-service attack.

See the information on ECom in the Symbian Developer Library as well as Writing Secure Servers in [Heath, 2006, Section 5].

Plug-ins

Ideally, plug-in providers shouldn't actually need to know how the IPC between the framework and cradle processes works. Indeed the less they need to know about the cradle processes the better. Unfortunately plug-in providers do need to know about the various cradle processes since during development they have to choose which one they would like to be loaded by according to which capabilities they need their plug-in to have at run time. This should be done by installing the two resource files needed by the framework along with the plug-in DLL as specified by the framework provider following the description given above.

Having done this, a plug-in developer should assign to the plug-in DLL the same capabilities as used by the cradle process it will be loaded by. This is done in the plug-in's MMP file using the `CAPABILITY` keyword as described in *Buckle* (see page 252).

Other than this, a plug-in should be implemented exactly as specified by the ECom service.

Consequences

Positives

- This pattern allows the plug-ins to run with capabilities independent of the framework process. This leads to both the framework and each plug-in running with the minimum capabilities that they individually and not collectively need. The main benefits of this are that:
 - The security risk to the device is reduced.
 - It is easier for developers to provide a plug-in.
- The security risk to the framework is reduced since the plug-ins run in a separate memory space.

- The security risk to the plug-ins is reduced since they run in a separate memory space from the framework and potentially from the other plug-ins as well.
- The capabilities of the framework can be increased without impacting the plug-ins, which makes maintenance easier.

Negatives

- An additional attack surface has been added which increases the security risk although this is mitigated by the fact that the security defense against this risk only needs to be implemented once by the framework provider unlike in *Quarantine* (see page 260), where each individual plug-in provider has to implement these defenses.
- If you host multiple plug-ins in the same cradle process then there is a risk that one plug-in will interfere with the other plug-ins whether by accident or design. For instance, if one plug-in panics then all the plug-ins in the same thread would be destroyed. Also, a plug-in would be able to interfere with the memory used by another plug-in and so change its behavior.
- Plug-ins do not have complete freedom to choose their own capabilities as in *Quarantine* (see page 260) but instead can choose from as many options as cradle EXEs are supplied by the framework provider.
- There is an additional performance overhead incurred by the creation of each cradle process (approximately 10 milliseconds longer⁴²) and by the use of IPC for each subsequent operation. If a significant amount of data needs to be transferred, then there could also be a sizeable performance overhead due to copying the data between the processes, although a data-sharing mechanism could be used to address this at the cost of further complicating the solution.
- Using an inter-process communication mechanism is more complex than the intra-process communication needed in *Buckle* (see page 252) which increases the development and maintenance costs of using this pattern.
- There is a default minimum of at least 21–34 KB RAM overhead⁴⁰ incurred by each cradle process created.
- Using *Client–Server* (see page 182) to implement this pattern would mean plug-ins also need to be assigned the `ProtServ` capability which increases the development costs of the plug-in providers slightly.

⁴⁰See Appendix A for more details.

Example Resolved

Framework

As described in the problem Example section, the SyncML subsystem needed to be extended to allow plug-ins to be provided that only needed to be assigned user-grantable capabilities whilst allowing other plug-ins to be provided that used additional system capabilities to execute their functionality.

One way that plug-in providers were freed from the need to have their plug-ins signed with system capabilities was to allow data provider plug-ins to no longer run within the Sync Agent process and instead to be hosted in a separate server process. Two standard cradle processes were added:

- Host Server 1 (smlhostserver1.exe) loads plug-ins which just need to be assigned the user-grantable capabilities ReadUserData and WriteUserData.
- Host Server 2 (smlhostserver2.exe) loads plug-ins which just need to be assigned the system capabilities ReadDeviceData and WriteDeviceData in addition to the user-grantable capabilities ReadUserData, WriteUserData and NetworkServices.

This is a good example of the Sync Agent process having capabilities independent of its plug-ins since it is assigned the system capability Prot-Serv in addition to the user-grantable capabilities NetworkServices, LocalServices, ReadUserData and WriteUserData.

This example chooses to implement the pattern using *Client-Server* (see page 182) for the IPC, which results in the structure shown in Figure 7.14, where just one of the host servers is shown.

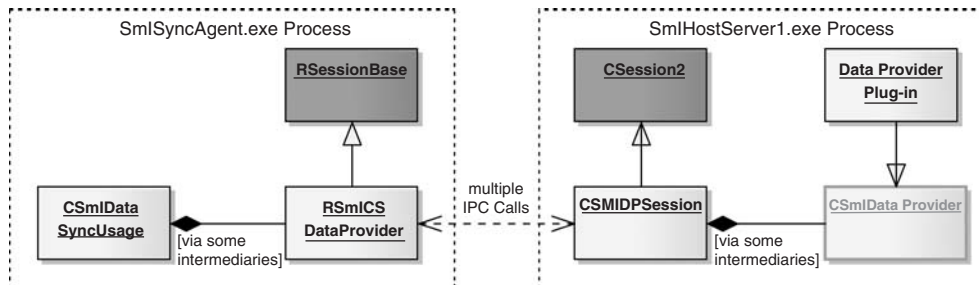


Figure 7.14 Structure of SyncML

The main components involved in this pattern are as follows:

- CSmlDataSyncUsage, which represents a data synchronization task, is the plug-in client.

- `RSmlCSDataProvider` is the framework proxy and implements the client session object.
- `CSmlDPSession` is the plug-in proxy and implements the server session object.
- `CSmlDataProvider` defines the plug-in interface that data provider plug-ins have to implement.

Not shown in Figure 7.14 is the `CSmlDPServer` class that derives from `CPolicyServer` rather than `CServer2` to allow a cradle process to check that each IPC message has come from the Sync Agent process by checking the caller's SID.

The diagram has been simplified by not showing some intermediate classes between `CSmlDataSyncUsage` and `RSmlCSDataProvider` and between `CSmlDPSession` and `CSmlDataProvider`; they are used to abstract away the details of whether a plug-in is loaded directly into the Sync Agent process or into a cradle process depending on what capabilities it requires.

Note that `ProtServ` is not used by `smlhostserver1.exe` since the whole point of this process is that it allows plug-ins to be loaded that only have to be assigned user-grantable capabilities which wouldn't be true if they also had to have `ProtServ`. This means a piece of malware could prevent them from being loaded. This is seen as a reasonable risk to take as those plug-ins are not essential to the working of the subsystem. Also, the denial-of-service risk can be relatively easily fixed by uninstalling the malware from any infected devices.

Plug-ins

The developer of a data provider plug-in has to provide two registration files:

- a file that specifies which cradle process should be used to load the plug-in, which is put into the Sync Agent's Import directory when the plug-in is installed
- a standard ECom resource file that allows a cradle process to load the plug-in via ECom.

For example, here is the registration file for the Bookmarks Data Provider:

```
// bookmarkdataproducerhs.rss

#include <DataProviderInfo.rh>
RESOURCE SYNCML_DATA_PROVIDER test
{
    id = 0x10009ECD; // Data Provider Identifier
}
```

```
version = 1;
display_name = "Bookmarks";
default_datastore_name = "bookmarks.dat";
supports_multiple_datastores = 0;
server_name = "SmlHostServer1";
server_image_name = "smlhostserver1";
mime_types =
{
    MIME_TYPE { mime="text/x-fl-bookmark"; version="1.0"; }
};
}
```

Other Known Uses

- *Device Management Agents*
Device management agents in the Device Provisioning subsystem also use this pattern to store remotely manageable device data. This pattern is used to allow third parties to supply some of the plug-ins. Originally *Buckle* (see page 252) was used to support device management adaptor plug-ins that model various types of device data, such as email accounts. However, a subsequent requirement to permit third parties to provide plug-ins meant that the existing need for plug-ins to have the NetworkControl capability would have incurred significant costs in time and money for this group of plug-in providers. Since ongoing communication was needed between the framework and the plug-ins, this pattern was chosen as the way to extend the subsystem to satisfy the new requirement.
- *Content Access Framework (CAF)*
CAF provides services that enable agents to publish content in a generic manner that is easy for applications to use. Users of CAF can access content in the same way regardless of whether the content is plain text, located in a server's private directory, or DRM protected. Those agents that don't require any capabilities can be loaded as in *Buckle* (see page 252). Since this isn't possible for DRM content, the architecture allows selected agents to be provided using this pattern.

Variants and Extensions

- *Combined Buckle and Cradle*
One variant of this pattern is where a framework chooses to load plug-ins either via *Buckle* (see page 252) or via *Cradle* (see page 273). This means that the framework loads plug-ins as DLLs into the framework process when a plug-in's capabilities are compatible with those of the framework. Plug-ins are only loaded by the framework

through a cradle process when the capabilities of the plug-in and the framework process are incompatible. This means the overhead of creating a new process for a plug-in is avoided unless absolutely necessary.

- *Buckle Converted into Cradle*
Here a framework simply loads plug-ins via *Buckle* (see page 252). However, if a plug-in finds that the limitations imposed by just running at the capabilities of the framework process prevent its operation then it may choose to supply an ECom plug-in that satisfies the framework, create a separate process and then forward all calls to this new process to be executed there instead. The framework is unaware that this is being done.

References

- Proxy [Gamma *et al.*, 1994], *Buckle* (see page 252) and *Client–Server* (see page 182) are normally used when implementing this pattern.
- *Buckle* (see page 252) is an alternative way of providing an extension point that is a simpler but less flexible solution to this problem.
- *Quarantine* (see page 260) is an alternative way of providing an extension point which doesn't require the plug-in to be signed with the same capabilities as the framework when you require little or no communication between the framework and its plug-ins.
- *Secure Agent* (see page 240) can be seen as the inverse of this pattern in that we are separating out the less trusted components into their own processes.
- See Chapter 3 for patterns describing how you can improve execution performance or RAM usage by changing when you load and unload plug-ins.
- Information on using Message Queues can be found in the Symbian Developer Library under Symbian OS guide » Base » Using User Library (E32) » Message Queue » Using Message Queue.
- See the following document for more information on resource files: Symbian Developer Library » Symbian OS guide » System Libraries » Using BAFL » Resource Files.

8

Optimizing Execution Time

This chapter is all about making the device appear faster to the end user. This is a separate idea from optimizing your component as that can also include reducing resource usage or drawing less power. This chapter is based on the recognition that it is pretty much impossible to create an optimal system in which every concern you might have has been addressed and its negative aspects minimized. You have to make trade-offs to optimize those attributes of your component that are most important. The patterns described in this chapter make the assumption that optimizing execution time is your top priority. Of course you will still be bound by factors such as limitations on the resources you use but they are assumed to be of secondary importance. So for instance, you might choose to introduce a cache¹ so that a frequently executed task can be done more quickly at the cost of increased RAM usage.

Another characteristic of optimizations is that they often only work in particular situations such as for a specific CPU architecture, a particular usage pattern or where there is high bandwidth available. This can make them fragile as, if the situation changes, the approach taken may not be the best choice, and in fact might make the task slower! Optimizations also introduce complexity as they tend to rely on special cases and therefore are likely to increase your maintenance costs. Hence a general recommendation is that you should resist the urge to optimize a component until you can prove that it is worth the effort to develop and the impacts it will have on your component. Indeed Donald Knuth said “We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.” [Knuth, 1974].

However, if you have measured your execution time and found it to be wanting, one way to optimize it is to simply do less work by getting rid of any wasted CPU cycles or by cutting back on some part of your activity. As an example of the latter, you might optimize the time taken to display an email inbox containing thousands of messages by only reading

¹ en.wikipedia.org/wiki/Cache.

from disk the messages that need to be shown in the view rather than all of them. There are a number of profiling tools that help you identify this kind of waste.

Once you've eliminated the more obvious waste, you need to start looking at other approaches. One tactic you might find useful is to 'fake' a decrease in execution time by simply re-arranging how you execute a task so that only those things that are essential to providing what the client or end user wants are done up front and then everything else is done as a background task. For instance, when an entry is deleted from a database, you don't need to compact it and clean up immediately so you could choose to mark the database field as empty and schedule a background activity to do the compaction after the client thinks his request has been completed. It's this latter strategy that *Episodes* (see page 289) focuses on by describing how a task can be cut up into episodes so that the critical ones required by the client are done first and their task reported as completed before all the non-critical tasks are performed.

The other pattern in this chapter, *Data Press* (see page 309), looks at the domain of middleware components that act as conduits for streams of data. The challenge here is to reduce the time taken to handle potentially large volumes of data since the main reason there is data to be handled in the first place is because some application has asked for it and wants to make use of it when it arrives. So we don't want to be wasting any time getting the data from place to place; we also want to reduce the potential for the data handling to fail. This is achieved by pre-allocating all the memory needed during the parsing of the stream and then using a long-lived object to interpret each packet of data in turn. This avoids time-consuming allocations that also introduce unwanted error paths.

Episodes

Intent Give the appearance of executing more swiftly by delaying all non-essential operations until after the main task has been completed.

AKA Staged Initialization, Phased Initialization, and Staged Start-up

Problem

Context

You need to perform a significant task on behalf of a client that can be broken down into a number of separate sequential episodes, only some of which need to be performed immediately.

Summary

- The client wants the task to appear to be completed as quickly as possible.
- It is important to you to be able to configure how the task is performed either now or in future.
- You want any optimizations to impact the maintainability of your component as little as possible.

Description

When you perform any non-trivial task, it will be formed of many individual parts that need to be completed. These might vary from something as simple as allocating an in-memory buffer to things as complex as checking the integrity of a database. Regardless of what these operations are, they each take some period of time to be performed.

This applies as much to applications interacting with the end user as to services even though the tasks they perform may differ. For example, an application needs to lay out UI controls on the screen, load images or load localized text whilst a service wouldn't need to do any these. However, the same general problem of a task for which many operations need to be performed is applicable to both situations. Some of the tasks which are applicable to both applications and services are allocation of memory, connecting to a service, requesting information from a service, reading data from disk, verifying data received and loading plug-ins. In many cases, the problem is recursive, with an application needing to connect to many servers as part of its task,

each of which need to perform their own tasks to satisfy the application's requests.

However, the key point is that often not all of the operations performed as part of a task are needed to satisfy the client's immediate request. Of course, sometimes this is because an operation simply isn't needed; as an example, consider an end user who views an email for the second time, using a messaging application. If your application goes off to the mail server to retrieve the message over the network both times then the second retrieval was a waste as the first copy of the message could simply have been stored locally. Such operations should be excised entirely and aren't the focus of this pattern.

This pattern focuses on operations which do need to be performed at some point but not immediately to satisfy the client's request. Usually such operations are for house-keeping purposes only needed by the component itself. For example, when removing data from a database, the data is first marked as deleted and then the database is compacted to actually remove it. Only the first operation needs to be performed as soon as possible. Once the data has been marked as deleted, the client can carry on while the non-essential compaction operation happens afterwards.

Note that this problem applies irrespective of whether the client is internal to your component or some external client such as the end user. In either case, you want to reduce the time taken to perform the task they've requested of your component.

Example

To illustrate this problem, consider the slightly simplified version of OggPlay [Wilden *et al.*, 2003] discussed here. This media-playing application performs the following high-level operations during its initialization task:

- read any data stored by the end user
- load the most recent track list shown to the end user
- start playing the last track selected by the end user
- construct and display the UI to the end user.

The final task effectively signals to the end user that the task of initializing the OggPlay application has completed by switching from its splash screen to its main screen as shown in Figure 8.1.

There are several approaches that we could take to performing these operations. The simplest approach would be to initialize everything together in one method. This would lead to something similar to the following code:



Figure 8.1 OggPlay screens: a) splash screen b) main application screen

```
void COggPlayApplication::ConstructL()
{
    // Load the end user's data
    iUserData = CUserData::NewL();
    TRequestStatus readingStatus;
    iUserData->ReadData(readingStatus);
    User::WaitForRequest(readingStatus); // Wait for completion
    User::LeaveIfError(readingStatus.Int());

    // Load the track list
    iTrackList = CTrackList::NewL();

    // Create the main view
    iMainView = CMainView::NewL();

    // Start last track playing
    iPlayer = CMusicPlayer::NewL();
    iPlayer->Play();

    // Initialization done, display main view
    iMainView->Display();
}
```

The method shown above has the advantages of being straightforward and containing all the code for the task in one location. This makes it easier to understand and easy to extend to add any extra operations in the future by just adding more lines of code to this method.

However, after some analysis we realize that some of the operations, such as reading in the end user's data from disk, take a long

time. As a consequence of this, we could decide to encapsulate each operation using *Active Objects* (see page 133) to allow multitasking of other operations while waiting for file IO to complete. However, since we need to perform the operations sequentially this isn't an option that is open to us. The requirement for sequential operations is usually to preserve the dependencies between the operations. In the code example above, playing the last track requires the track list to be loaded first.

Even if the operations could be parallelized there can be the danger that you over-parallelize them. Occasionally the overhead of doing this actually makes the whole task take longer. For instance, if the individual operations are requests to services across the system then you can end up in a situation in which more time is wasted context switching between lots of different threads than is saved by multitasking.

A third approach would be to change each operation so that it is only done on demand. For instance, only loading the track list when some information about the track list is requested.² This initially might seem like the ideal option as it allows the application to do the necessary initialization just when it's needed. However, this approach is not without its flaws. Firstly, it increases the complexity of your design, especially if lots of operations are involved. Secondly, if you know the operation is needed then you may make the responsiveness worse elsewhere. In this case, the end user waits longer than usual when they first try to access the track list. Finally, you lose control of the task with the ordering of operations not being decided during development but by run-time choices and dependencies. This makes debugging any issues significantly more problematic by making behavior difficult to predict and hence increasing maintenance and test costs.

Solution

The solution is to encapsulate a group of operations as an *episode*. As a minimum you need to have a required episode, after which the client is told that their requested task has completed, and a background episode in which the house-keeping tasks can be done in a more relaxed manner without a fixed deadline to meet. You may find that it makes your design or maintenance easier to have more than two episodes.

The episodes are then executed sequentially by an object, called here a *serializer*, to give you predictable behavior. This allows you to specify the exact order in which operations need to occur to achieve the task but by encapsulating this you have an easy way to manipulate what episodes occur during the task. The understanding and control that this gives you allows you to ensure the task is not only done correctly but done as efficiently as possible.

²Further details of this approach are covered in *Lazy Allocation* (see page 63).

Structure

The client is responsible for starting the whole task by creating the serializer (see Figure 8.2). The client uses *Event Mixin* (see page 93) to register with the serializer for notification of when the task has been completed at some point later.

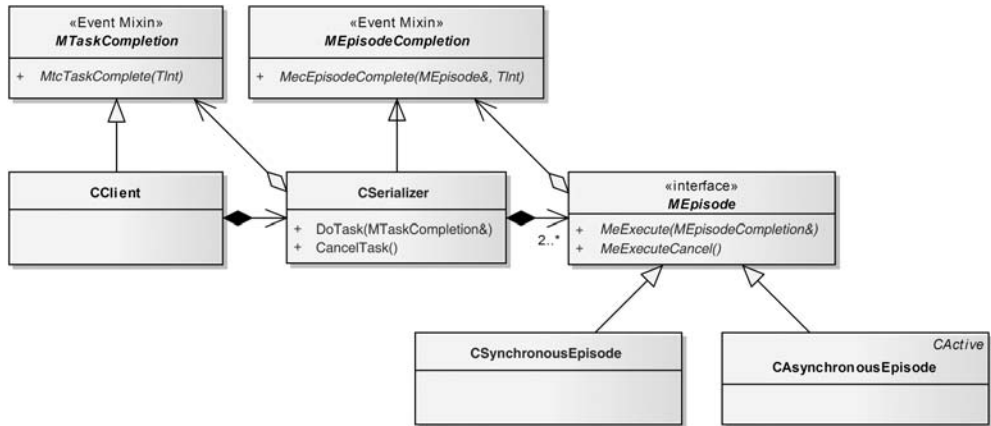


Figure 8.2 Structure of the *Episodes* pattern

The serializer is responsible for creating and managing each of the episodes required for the overall task. It does this by managing a list of the episodes which it goes through in order, executing each one fully before moving onto the next. This object uses *Event Mixin* (see page 93) to register with each episode for notification of when it has completed its work. Note that there must be at least two episodes owned by the serializer since there are, at least, a required and a background episode.

Each episode derives from a common interface, **MEpisode**, so that the serializer can be fully decoupled from each of the episodes. The interface is designed to allow episodes to execute both synchronously and asynchronously. Episodes that you wish to work asynchronously should use *Active Objects* (see page 133) to achieve this.

Dynamics

The dynamics is illustrated in Figure 8.3, which shows three episodes, two required and one background episode. For simplicity, the creation of all the episodes is not shown. However, we assume here that all the episodes are created by the serializer when it is created by the client.

This sequence shows the serializer simply traversing through its list of episodes executing them one by one. Once the last required episode has completed, it notifies the client that the task is done. After the task is completed, the background episode is executed.

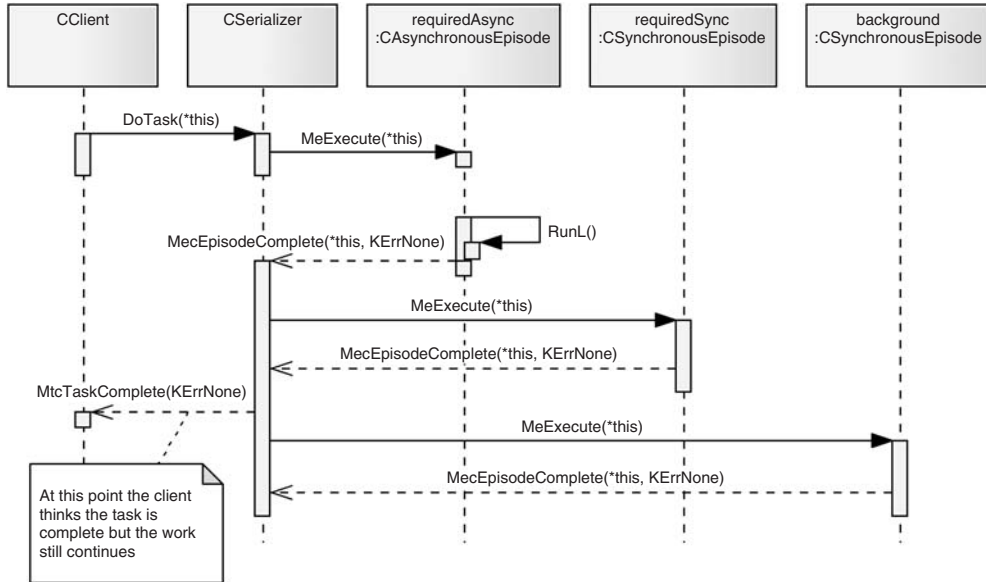


Figure 8.3 Dynamics of the *Episodes* pattern

If an error occurs during the execution of an individual episode, this should result in the episode calling `MecEpisodeComplete()` to pass the error code on to the serializer to handle. Normally, the serializer would then halt its progression through the episodes and call `MtcTaskComplete()` on the client and pass on the error from the episode that failed. Note that neither of these two completion functions should `Leave`. This is because doing so would pass the error away from the object that has responsibility for handling the error which is one of the consequences of using an asynchronous API here.

Implementation

Before getting into further details of how to construct each of the objects involved, you need to decide how to sub-divide the overall task into episodes. As a first step you need to identify each of the individual operations involved and the dependencies between them. This allows you to identify which are required before the client can be told the task has been completed and which can be done afterwards. You could stop at this point with two episodes however it is often useful to sub-divide the operations a little more. The reason for this is that the more episodes you have, the more flexible and maintainable your component will be in future since it is relatively easy to re-order episodes or insert new ones

whereas moving an operation from one episode to another is usually more tricky since closely related operations are normally encapsulated in the same episode. The downside to having more episodes is that the overhead of this pattern increases so you need to balance how much you optimize the task against flexibility for future changes.

Serializer

The serializer would be defined as follows:

```
class MEpisodeCompletion
{
public:
    virtual void MecEpisodeComplete(MEpisode& aEpisode, TInt aError) = 0;
};

class CSerializer : public CBase, public MEpisodeCompletion
{
public:
    CSerializer* NewL();

    void DoTask(MTaskCompletion& aTaskObserver);
    void CancelTask();

    // From MEpisodeCompletion
    virtual void MecEpisodeComplete(MEpisode& aEpisode, TInt aError);
private:
    CSerializer();
    void ConstructL();
    void NotifyTaskComplete(TInt aError);
private:
    MTaskCompletion* iTaskObserver;
    TInt iFirstBackgroundEpisode;
    RPointerArray<MEpisode> iEpisodes;
    TInt iCurrentEpisode;
};
```

The CSerializer class contains the following member variables:

- iTaskObserver, which allows the task to be completed
- iFirstBackgroundEpisode, which specifies when the client should be informed the task has been completed
- iEpisodes, the array of episodes to execute
- iCurrentEpisode, the current index to iEpisodes.

Implementations of the most interesting methods in CSerializer follow. The first is the ConstructL() method, in which all the episodes are created and added to the iEpisodes array. In this method, you should order the episodes as necessary to get the task done. The only

other thing you need to do is mark the point at which the client should be notified that the task is done by setting `iFirstBackgroundEpisode`.

```
void CSerializer::ConstructL()
{
    CRequiredAsyncEpisode* ep1 = CRequiredAsyncEpisode::NewLC();
    iEpisodes.AppendL(ep1);
    CleanupStack::Pop(ep1);

    CRequiredSyncEpisode* ep2 = CRequiredSyncEpisode::NewLC();
    iEpisodes.AppendL(ep2);
    CleanupStack::Pop(ep2);

    // Count is used rather than a hard-coded value so that you don't have
    // to remember to change the value if you add more episodes above
    iFirstBackgroundEpisode = iEpisodes.Count();

    CBackgroundSyncEpisode* ep3 = CBackgroundSyncEpisode::NewLC();
    iEpisodes.AppendL(ep3);
    CleanupStack::Pop(ep3);
}
```

The following method is used by the client to start the task. Note that this function uses *Escalate Errors* (see page 32) to resolve errors but it doesn't do so using the Leave mechanism. Instead all errors are passed up to the client through the `MtcTaskComplete()` function so that the client only has the one error path to deal with.

The other important point about this function is that you need to check whether or not there is already a task ongoing but this can get quite tricky if the required episodes have already been done since the client would be within its rights to think it could ask for the task to be performed again. If this is the case then you have to decide how to deal with this. The code below deals with it very simply by sending an error to the client but this isn't very user friendly! The best way to resolve the situation depends on what your background episodes do. For instance, it might be that you can simply cancel the background task and allow the new request to take precedence. Alternatively, you might need to accept the new task and perform it at the same time as the background episodes from the previous task by using *Active Objects* (see page 133) specifically to do those tasks and then dying.

```
void CSerializer::DoTask(MTaskCompletion& aTaskObserver)
{
    // Check we're not already busy with a previous task
    if (iTaskObserver)
    {
        aTaskObserver.MtcTaskComplete(KErrInUse);
    }
}
```



```

// Check to see if the background episodes of the previous task are
// still being performed
if (iCurrentEpisode < iEpisodes.Count())
{
    aTaskObserver.MtcTaskComplete(KErrInUse); // Or something else ...
}
else
{
    // Need to remember this for the completion later
    iTaskObserver = &aTaskObserver;

    // Reset the episodes as appropriate

    // Start at first episode
    iCurrentEpisode = 0;
    iEpisode[iCurrentEpisode]->MeExecute(*this);
}
}

```

The `MecEpisodeComplete()` function is where most of the work goes on by dealing with one episode completing and then going on to the next if there is one:

```

void CSerializer::MecEpisodeComplete(MEpisode& aEpisode, TInt aError)
{
    ASSERT(aEpisode == iEpisode[iCurrentEpisode]);

    if(aError)
    {
        // Handle the error from a failed episode by passing it to
        // the client to resolve
        NotifyTaskComplete(aError);
        return;
    }

    // Move to next episode
    ++iCurrentEpisode;
    if(iCurrentEpisode == iFirstBackgroundEpisode)
    {
        // Finished executing the required episodes so signal success
        // to the client
        NotifyTaskComplete(KErrNone);
    }

    if (iCurrentEpisode < iEpisodes.Count())
    {
        // Execute the next episode
        iEpisode[iCurrentEpisode]->MeExecute(*this);
    }
}

```

The `NotifyTaskComplete()` function just helps ensure that we maintain the following class invariant: that `iTaskObserver` is not `NULL`

only when the required episodes of a task are being executed:

```
void CSerializer::NotifyTaskComplete(TInt aError)
{
    iTaskObserver->MtcTaskComplete(aError);
    iTaskObserver = NULL;
}
```

In the following code, we implement the function to cancel the task. Note that it doesn't really make sense to allow a client to cancel the background episodes since it shouldn't even be aware that they're being executed. Hence we should only act on the cancel if the required episodes are still being executed.

```
void CSerializer::CancelTask()
{
    // Check we've been called with a task actually ongoing
    if (iTaskObserver)
    {
        iEpisode[iCurrentEpisode]->MeExecuteCancel();
    }
}
```

Episodes

All episodes derive from the following interface that is implemented as an M class:

```
class MEpisode
{
public:
    virtual void MeExecute(MEpisodeCompletion& aEpisodeObserver) = 0;
    virtual void MeExecuteCancel() = 0;
};
```

We show the implementation of the required synchronous episode first because it's easy to understand. Note that the synchronous background operation isn't shown because it'll be implemented in just the same way as this episode.

```
class CRequiredSyncEpisode: public CBase, public MEpisode
{
public:
    static CRequiredSyncEpisode* NewLC();

public: // From MEpisode
```

```
virtual void MeExecute(MEpisodeCompletion& aEpisodeObserver);
virtual void MeExecuteCancel();
...
};
```

The `MeExecute()` function is conceptually very simple. All it does is synchronously perform the operations that have been grouped into this episode and signal to the serializer when it's done. Note that this function uses *Escalate Errors* (see page 32) to resolve errors but it doesn't do so using the Leave mechanism. Instead all errors are passed up to the client through the `MecEpisodeComplete()` function so that the serializer only has the one error path to deal with.

```
void CRequiredSyncEpisode::MeExecute(MEpisodeCompletion& aEpisodeObserver)
{
    TInt err = KErrNone;

    // Perform synchronous actions

    // Complete client immediately
    aEpisodeObserver.MecEpisodeComplete(*this, err);
}
```

The implementation of `MeExecuteCancel()` does not need to perform any actions as there is no opportunity to cancel this episode:

```
void CRequiredSyncEpisode::MeExecuteCancel()
{
}
```

Here's the definition of an episode that derives from `CActive` to allow it to perform its execution asynchronously:

```
class CRequiredAsyncEpisode : public CActive, public MEpisode
{
public:
    static CRequiredAsyncEpisode* NewLC();
public: // From MEpisode
    virtual void MeExecute(MEpisodeCompletion& aEpisodeObserver);
    virtual void MeExecuteCancel();
public: // From CActive
    virtual void RunL();
    virtual TInt RunError(TInt aError);
    virtual void DoCancel();
private:
    void NotifyEpisodeComplete(TInt aError);
    ...
private:
    MEpisodeCompletion* iEpisodeObserver;
};
```

Note that the active object priority given to an asynchronous episode depends on whether it's a required episode, when you'd expect it to have a high priority, or whether it's a background episode, when you'd expect it to have a low priority perhaps even `EPriorityIdle` so it only runs when the thread has nothing else to do.

The asynchronous implementation of the `MeExecute()` method needs to remember the `aEpisodeObserver` passed as a parameter to allow it to be completed once the episode has finished. Other than that it simply needs to perform its asynchronous operation:

```
void CRequiredAsyncEpisode::MeExecute(MEpisodeCompletion&
                                     aEpisodeObserver)
{
    iEpisodeObserver = &aEpisodeObserver;
    IssuesAsyncRequest(iStatus);
    SetActive();
}
```

Once the asynchronous request has completed, `RunL()` is called to handle the result. This might be to perform further synchronous operations or simply to notify the completion of the episode:

```
void CRequiredAsyncEpisode::RunL()
{
    User::LeaveIfError(iStatus);

    // Perform further synchronous operations as needed
    TInt err = KErrNone;
    ...

    // Operations are all done so tell the serializer
    NotifyEpisodeComplete(err);
}
```

We use the `RunError()` method to provide our error handling which is simply to notify the serializer that the episode has completed though you could try to resolve the error by retrying an operation, for example, before handing the error on.

```
TInt CRequiredAsyncEpisode::RunError(TInt aError)
{
    NotifyEpisodeComplete(aError);
    return KErrNone;
}
```

The `NotifyEpisodeComplete()` function just helps ensure that we maintain the following class invariant: that `iEpisodeObserver` is not `NULL` only when the episode is executing its operations:

```
void CRequiredAsyncEpisode::NotifyEpisodeComplete(TInt aError)
{
    iEpisodeObserver->MecEpisodeComplete(*this, aError);
    iEpisodeObserver = NULL;
}
```

Finally, we need to ensure that the serializer can correctly cancel the episode. We assume that you will also provide an implementation of your `DoCancel()` function³ to cancel your implementation-specific asynchronous operation.

```
void CRequiredAsyncEpisode::MeExecuteCancel()
{
    // The active object cancel is used to cancel the current asynchronous
    // operation that the episode is performing
    Cancel();

    // Notify the serializer that the cancel is complete
    if (iEpisodeObserver)
    {
        NotifyEpisodeComplete(KErrCancel);
    }
}
```

Consequences

Positives

- The task is executed more quickly because all of the operations not immediately necessary to complete the task are delayed until after the task appears to have completed.
- The task is easier to maintain since each episode is treated in exactly the same way and there is no possibility of treating one episode differently from another which might otherwise catch out future developers of your component. One example of this is the way that synchronous and asynchronous episodes appear to work in the same way to the serializer which allows you to easily convert episodes between these two forms of execution. In addition, the layout of the task is very clear, which helps maintainers to understand what it is doing.
- This solution increases the flexibility you have in changing the order in which you perform episodes by having a single place in which their order is set up. This allows you to easily change the order or even add further episodes in future. Dependencies between the episodes are easily controlled and are more likely to be respected correctly because they are executed in a linear order and do not interfere with each other.

³Required as part of `CRequiredAsyncEpisode` being an active object.

Negatives

- Additional development time is needed upfront to correctly understand the way the task operates and how it might be divided into episodes.
- You introduce a potentially complicated situation in which a client may request a new task to be performed whilst the background episodes of the previous task are still executing. For some tasks, this can be a significant problem that needs to be solved.
- Circular dependencies between episodes can't be expressed by the linear ordering of them imposed by the serializer. This pattern is only able to handle circular dependencies by encapsulating them within a single episode. Assuming, of course, that you can't find some way to break the circular dependency altogether.
- Episodes cannot be immediately parallelized which might prevent further execution time optimizations. However, if you do need to parallelize episodes then this pattern provides a starting point for a design that can cope with this.
- This solution adds some overhead to the processing of the task. This is normally a relatively small number of additional CPU cycles as well as code size to organize the episodes. For a non-trivial overall task divided up into a small number of episodes, this isn't usually significant.

Example Resolved

Here, we consider the simplified problem described earlier to avoid any unrelated complexity.⁴ In applying this pattern to the media player initialization, the first decision is to choose the episodes; since the initialization of the application splits up nicely into four top-level operations, we use them as our episodes:

- reading data stored by the end user
- loading the most recent track list shown to the end user
- playing the last track selected by the end user
- constructing and displaying the UI to the end user.

If we were to simply execute them in the order above, we wouldn't see any optimization so we choose to delay the playing of the last track selected by the end user until after we've shown the UI to the end user.

⁴For the complete solution in OggPlay, please look at the source code given by [Wilden *et al.*, 2003].

Unfortunately, we can't delay the other episodes because they retrieve information used when displaying the UI.

Having chosen the episodes and their order, we represent this in the code within the following function. Note that this function is called when `COggPlaySerializer` is created by `COggPlayApplication`:

```
void COggPlaySerializer::ConstructL()
{
    CUserDataEpisode* ude = CUserDataEpisode::NewLC();
    iEpisodes.AppendL(ude);
    CleanupStack::Pop(ude);

    CTrackListEpisode* tle = CTrackListEpisode::NewLC();
    iEpisodes.AppendL(tle);
    CleanupStack::Pop(tle);

    CMainViewEpisode* mve = CMainViewEpisode::NewLC();
    iEpisodes.AppendL(mve);
    CleanupStack::Pop(mve);

    iFirstBackgroundEpisode = iEpisodes.Count();

    CMediaPlayerEpisode* mpe = CMediaPlayerEpisode::NewLC();
    iEpisodes.AppendL(mpe);
    CleanupStack::Pop(mpe);
}
```

The `COggPlaySerializer` class is otherwise constructed exactly as given in the pattern above.

When constructing the individual episodes we need to take into account whether or not they need to be performed synchronously or asynchronously. In the example here, only the reading in of the end user data needs to be done asynchronously so this is the only episode that is implemented as an active object.

The following code shows possible implementations for the `Execute()` methods for the four episodes described above. The implementation of the register methods such as `RegisterUserData()` are not shown but would be responsible for transferring ownership of the created objects to the appropriate part of the application.

```
void CUserDataEpisode::Execute(MEpisodeCompletion& aEpisodeObserver)
{
    // Remember the requesting callback to complete later on
    iEpisodeObserver = aEpisodeObserver;

    // Load end user data
    TRAPD(err, iUserData = CUserData::NewL());
    if (err == KErrNone)
    {
        iUserData->ReadData(iStatus);
    }
}
```

```

        SetActive();
    }
    else
    {
        // An error occurred, so end the episode immediately
        NotifyEpisodeComplete(err);
    }
}

// since CUserDataEpisode derives from CActive
void CUserDataEpisode::RunL()
{
    User::LeaveIfError(iStatus);

    // Success so transfer ownership of end user data
    RegisterUserData(userData);

    // End the episode successfully
    NotifyEpisodeComplete(iStatus.Int());
}

void CTrackListEpisode::Execute(MEpisodeCompletion& aEpisodeObserver)
{
    // Load the track list
    CTrackList* trackList = NULL;
    TRAPD(err, trackList = CTrackList::NewL());
    if(err == KErrNone)
    {
        RegisterTrackList(trackList); // Transfer ownership of track list
    }

    // End the episode synchronously
    aEpisodeObserver->EpisodeComplete(*this, err);
}

void CMainViewEpisode::Execute(MEpisodeCompletion& aEpisodeObserver)
{
    // Create the main view
    CMainView* mainView = NULL;
    TRAPD(err, mainView = CMainView::NewL());
    if(err == KErrNone)
    {
        mainView->Display(); // Initialization done so display the main view
        RegisterMainView(mainView); // Transfer ownership of the main view
    }

    // End the episode synchronously
    aEpisodeObserver->EpisodeComplete(*this, err);
}

void CMusicPlayerEpisode::Execute(MEpisodeCompletion& aEpisodeObserver)
{
    CMusicPlayer* musicPlayer = NULL;
    TRAPD(err, musicPlayer = CMusicPlayer::NewL());
    if(err == KErrNone)
    {
        musicPlayer->Play();
    }
}

```



```
// Transfer ownership of the music player
RegisterMusicPlayer(musicPlayer);
}

// End the episode synchronously
aEpisodeObserver->EpisodeComplete(*this, err);
}
```

Other Known Uses

- *Application Splash Screens*

It is common practice for applications to use a splash screen in the same way as used by OggPlay to give the end user near instant feedback that the application is loading. This makes the time the end user waits for the application to initialize less frustrating as they are not left wondering if they really pressed the launch button. The episodes used by such applications can be roughly categorized as:

- Task 1 – React to button press
 - Creation of the application process and main objects
 - Show the splash screen (*task complete*)
- Task 2 – Display interactive UI
 - Create UI controls, etc.
 - Switch splash screen for the main UI (*task complete*)
 - Background operations

The two tasks are normally run back-to-back using a single serializer.

- *Loading Protocol Plug-ins*

The Communications Infrastructure subsystem of Symbian OS uses this pattern to load different protocol plug-ins at different stages of device start-up. An episode is used to encapsulate each stage of device start-up and they are implemented using *Active Objects* (see page 133) to allow them to asynchronously wait for notification of the next state change.⁵ When notification is received, the episode is responsible for loading the associated group of protocols.

This allows protocols to be loaded in stages so that only those protocols required to show the main device UI are loaded early. The loading of non-essential or background protocols is deferred until later in the start-up process after the end user thinks the device has booted.

⁵Distributed by the System Starter using *Coordinator* (see page 211).

- *After-Market Application Starter*
This component introduced into Symbian OS v9.5 allows applications that have been installed after the device has been created, to register to be started when the device starts up. Each application to be started is represented by an individual episode. Note that this is an example where the pattern is not primarily used to optimize execution time but rather to allow the order of the list of applications to start to be controlled dynamically.

Variants and Extensions

- *Long-Running Serializer*
If your task takes a long time to complete, then you'll find that your thread is less responsive in handling events such those generated by the end user interacting with your UI. This is because the pattern as implemented above synchronously traverses the list of episodes and hence doesn't provide points where it yields to other tasks being multitasked in your thread.
If this is a problem then you can solve it by making the serializer a 'long-running active object' as described in *Asynchronous Controller* (see page 148) though this will be at the cost of slightly increased overhead in terms of both CPU cycles and code size.
- *Self-Determining Background Episode*
One way to solve the problem of a client requesting a new task to be executed before the background episodes from a previous task have been completed is for you to encapsulate all the background operations into a single episode. When the serializer comes to notify the client that the task is complete it tells the background episode to start executing and then passes ownership of the episode to the episode itself. The serializer is then free to accept the next request from a client for it to perform a task. The background episode then executes as normal but when it's done it destroys itself rather than notifying the serializer.
- *Dynamic Configuration of Episodes*
Storing the individual episodes as an array of pointers to a common interface allows the order of the episodes to be easily changed or even for episodes to be removed. This variant takes this a step further to prioritize or eliminate episodes at run time to allow the component to be easily configured for a particular environment.
The optional inclusion of episodes can be accomplished easily with code similar to the following in the construction of the serializer where the task configuration is read in from an external source

such as a resource file or the Feature Manager⁶ service provided by Symbian OS.

```
void CSerializer::ConstructL()
{
    // Read task configuration
    TTaskConfiguration configuration = ReadTaskConfiguration();

    // Create the initial episodes

    // Optionally include the network connection episode
    if(configuration.NetworkConnectionRequired())
    {
        CNetworkEpisode* ne = CNetworkEpisode::NewLC();
        iEpisodes.AppendL(ne);
        CleanupStack::Pop(ne);
    }
}
```

You can also dynamically change the ordering of episodes at run time as follows:

```
void CSerializer::ConstructL()
{
    // Read task configuration
    TTaskConfiguration configuration = ReadTaskConfiguration();

    // Create the initial episodes

    // Create the cache initialization episode
    CCacheEpisode* ce = CCacheEpisode::NewLC();

    // Optionally add the cache episode as the first episode
    if(configuration.InitializeCacheFirst())
    {
        iEpisodes.InsertL(ce, 0); // Insert as first episode
    }
    else
    {
        iEpisodes.AppendL(ce); // Add to the end of the episode list
    }
    CleanupStack::Pop(ce);
}
```

- *Parallelizing Episodes*

The main version of this pattern describes episodes that are started in sequence. However, this same design can be extended to allow episodes to be started in parallel. This is usually done by allowing

⁶Known as the Feature Registry in some of the early versions of Symbian OS v9.

each episode to specify which of the following behaviors should be used when the serializer executes the episode:

- **Fire and forget** – the serializer starts an episode and then immediately moves on to the next in the sequence without waiting for notification that the episode has finished executing.
- **Wait** – the serializer starts an episode and waits for notification that it has finished executing. If all episodes use this behavior then you get the same behavior as in the main pattern.
- **Deferred Wait** – the serializer starts an episode and immediately goes on to start the next episode. The serializer will go on starting episodes until it comes to a synchronization point, often implemented as an episode. The serializer then waits for all deferred wait episodes to finish executing before moving past the synchronization point to the subsequent episode.

As you can imagine, these behaviors make the serializer significantly more complicated as well as making it more difficult to decide upon an optimum way of executing the episodes. However, it does give the opportunity to further optimize execution time by parallelizing the episodes. Of course, this isn't a guarantee of reducing execution time since you could waste more time switching context between lots of different threads if the individual operations of the overall tasks are mainly requests to services across the system.

References

- *Active Objects* (see page 133) is used to implement asynchronous episodes.
- *Asynchronous Controller* (see page 148) can be used to prevent this pattern impacting the responsiveness of your thread.
- *Coordinator* (see page 211) combines well with this pattern by having an asynchronous episode act as a responder so that it receives notifications of the coordinated state change. On receiving a notification, it performs the next step in the overall task.

Data Press

Intent Quickly process an incoming stream of data packets, without allocating any further memory, by using a long-lived object to interpret each packet in turn.

AKA None known

Problem

Context

You need to efficiently parse a stream of many similar blocks of data, known as *packets*,⁷ to route the data stream to its correct destination.

Summary

- You need to minimize the time between receiving a packet and passing it on to its destination, which is known as the *latency*⁸ of your component.
- Your component needs to use as little resource, such as RAM, as possible, irrespective of how much data it processes, to allow the consumer of the data stream as much freedom to utilize it as possible.
- You wish to reduce the power usage of your component as it handles large volumes of data.
- Your parsing algorithms need to be flexible and easily maintained to cope with future upgrades and improvements.

Description

Data streams are divided up into packets to allow the underlying communication channel to be more easily shared between each of its users. This is because a packet contains a *header*⁹ consisting of control information such as the source and destination addresses, error detection codes such as checksums, and sequencing information. The rest of the packet is known as the *payload* which is the data that the user wishes to send.

Processing streams of data formed as a series of packets is a very common scenario in a protocol stack that manages the exchange of data

⁷[en.wikipedia.org/wiki/Packet_\(information_technology\)](https://en.wikipedia.org/wiki/Packet_(information_technology)).

⁸[en.wikipedia.org/wiki/Latency_\(engineering\)](https://en.wikipedia.org/wiki/Latency_(engineering)).

⁹Some packets contain a trailer at the end of the packet instead of at the front. However, this doesn't significantly impact the design so we won't go into any detail for this variation.

between two devices or across a network and hence this problem is most commonly seen by device creators. However, since streaming or frequent message passing can also occur within a device, or even between different classes within one process, this problem is also occasionally seen by application developers.

In each of these scenarios, there may be several layers of middleware protocols that handle the data in transit, with each layer often having to do quite a lot of work in addition to ensuring the data reaches the correct destination. For instance, it may need to be decrypted or uncompressed before being passed on.

Still, the most important part of each packet is the payload that is destined for an application or end user. Middleware should ideally operate as transparently as possible, processing each packet quickly and efficiently to leave plenty of CPU and RAM available for processing the payload when it reaches its destination. For instance, if the data is a high-quality video stream for the end user then the device should focus as much as possible of its resources on displaying that content to the end user rather than simply downloading it. After all, your parsing algorithm is just a piece of middleware. The interesting data is inside the packet you are processing and it would be beneficial for this data to be received as soon as possible.

Sometimes there are strict real-time constraints on the delivery of streamed data that make a low-latency parsing algorithm particularly important. Packets carrying voice data need to reach their destination within perhaps 10 or 20 ms to provide a high-quality communication link.

Another factor to consider is that it's no good processing the packets quickly if you drain the battery in doing so. As with any software on a mobile device, conserving battery power is an important consideration especially if you will be processing large volumes of data packets. Reducing the number of CPU cycles needed to parse each packet not only reduces the latency of your middleware component but also extends battery life. RAM efficiency can also help to conserve battery life by allowing RAM chips to be powered down or by indirectly reducing the CPU cycles used. Note that since your component will probably handle large volumes of data there is the potential for a mistake here to have a large impact on RAM usage.

In your packet-processing scenario, you'll find that there are a number of common factors that you can take advantage of to model the data and your functionality in an organized way to make your code manageable and maintainable. Normally each packet has a well-known structure, perhaps specified in an industry standard, and your middleware component will need to perform one of a standard set of operations on it. These operations are normally very similar each time a new packet of data is

processed. However, you need to design in the expectation that the rules and standards governing the behavior of your middleware component could be changed or extended in future.

Among the basic questions that will guide the design of the parsing algorithm for a middleware component are these:

- What is the format of the packet?
- What will the parser change in the packet?
- How will the parser change the packet?

Example

A number of examples of this problem come from individual protocol modules needing to process data packets received by the Symbian OS Communications Infrastructure. In this situation, when a data packet is prepared for transmission, it is typically structured so there is an informational header at the start of the packet and then the payload data. Hence, as a packet received from a remote device will have been processed by a whole stack of communication protocols, it will have a stack of headers above the payload data.

As a received packet is processed through the inbound stack, each protocol identifies and analyzes the header set by its remote peer protocol; parses the data it finds there and then, if all is well, passes the data packet on up the stack, usually removing the header it has just parsed. As such it is common for the start of the data packet to get closer to the payload at each higher layer in the stack.

For this specific example, we focus on just one of the protocols in the Symbian OS Bluetooth protocol stack – the Audio/Video Distribution Transport Protocol.¹⁰ This protocol allows audio and video data to be streamed between two devices such as a mobile device acting as a music player and a set of headphones. To do this, the protocol specification defines two separate channels:

- a signaling channel for A/V stream negotiation and establishment
- a transmission channel to carry the actual A/V data.

AVDTP sits at the top of a protocol stack and receives data from the L2CAP Bluetooth protocol below which needs to be passed onto the client above.

The algorithm for parsing each packet received on the signaling channel needs to first identify the protocol header and then parse and analyze it. The information in the header triggers state changes in which other protocol tasks are performed, such as starting or stopping the

¹⁰bluetooth.com/Bluetooth/Technology/Works/AVDTP.htm.

transmission channel. In this example, the payloads of the packets are fairly small and, in some cases, not present so no transformations of the data are needed before passing the packet on up the protocol stack.

Solution

An efficient solution for data parsing can be achieved using a pattern that is analogous to processing sheet metal in a machine press.¹¹ The machine stamps or molds each raw sheet of metal into a particular form according to whatever stencil¹² has been attached to the press tool. The stencil can be changed to produce a different effect on the metal – perhaps bending, trimming or punching holes in it. The same press and stencil combination may well process a number of raw sheets of metal in a continuous sequence.

As you can see this is a concept that resonates for our problem as follows:

- Each packet is formed of a section of *raw data*.
- The well-known format of the packets is a *stencil*.
- The parsing operations are performed using a stencil by the *data press*.

Our solution can then be described as follows: the raw data is passed from a *source* to a class that encapsulates the data press operations. This data press passes the raw data to a stencil class whose sole responsibility is to interpret a packet's worth of data and then inform the data press of what the packet means. It is the responsibility of the data press to act on this information by, for instance, passing the data on to the destination, or *sink*, of the data. The stencil object is then re-used to interpret the next packet's worth of data until the stream has been consumed.

Structure

The source component from which the data press receives raw data and the sink component to which it routes data are usually already present, or at least defined, when this pattern is used to develop a middleware component so this pattern doesn't focus much attention on these except for how they interact with the other objects. Note that the `RawData` class is intended to represent some storage class such as a C class, a descriptor or perhaps an `RMBufChain`.

The solution is expressed by the structural diagram in Figure 8.4. The data press itself provides the main interface to the middleware component as a whole and is alive the entire time the middleware component is being used to process a stream of data. It is responsible for creating the

¹¹en.wikipedia.org/wiki/Machine_press.

¹²en.wikipedia.org/wiki/Stencil.

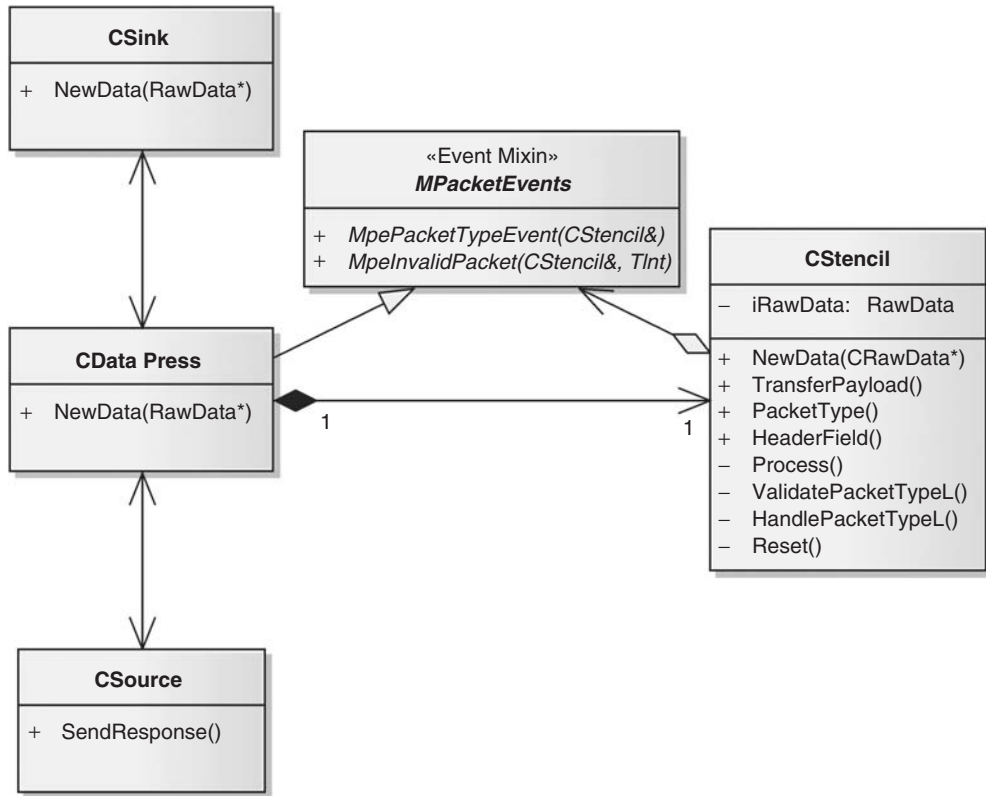


Figure 8.4 Structure of the *Data Press* pattern

other objects it depends on, such as the stencil. There are a number of choices for how this is done including the patterns described in Chapter 3. However, a common choice for the lifetime of the stencil object is *Immortal* (see page 53) so that it is created and destroyed with the data press object. This is in the expectation that the stencil will always be needed by the data press object as no data can be routed to its destination without first being interpreted by the stencil.

This approach neatly avoids the need to allocate an object simply to parse the incoming data. This is a significant benefit, as allocations are normally costly and unbounded operations would otherwise negatively impact your latency and perhaps break any real-time guarantees your component has promised to maintain. A final consideration is that by doing this you should be able to avoid any system errors whilst parsing inbound data which will make your design more robust. Of course you'll still need to handle domain errors such as malformed packets.

The data press uses *Event Mixin* (see page 93) to provide the stencil with an interface through which it can pass the results of its interpretation

of each packet. Commonly, this interface defines a method for each type of packet that the raw data can be interpreted as, in addition to a function such as `MpeInvalidPacket()` that is used to tell the data press that the raw data is corrupt in some way.

The stencil is used to encapsulate the raw data in terms of a well-defined format. As such, the stencil models the header of the packet and provides the data press with accessor functions for the various individual pieces of information in the header, also known as *fields*. These accessor functions interpret the raw data on demand and return the required value. By accessing fields through these functions the caller can be entirely unaware of how the header field is actually represented in the raw data. For instance, this allows you to confine the awareness of how the raw data is ordered, a concept known as *endianess*,¹³ to just the implementation of these functions.

The stencil relies on the persistence of the raw data all the while it is processing it. In addition, it must have sole access to the data so that you do not need to have the overhead of synchronizing access to the data to avoid corrupting it.

Figure 8.4 assumes that only one type of stencil is needed in your component. This is perfectly OK if each of the different packet types you might receive have a fairly similar structure. In this case, a single stencil class is appropriate and it will just have distinct functions to validate and handle each specific packet type. However, you might find that a group of distinct stencil types is more appropriate when you need to parse a diverse set of packet formats.

Dynamics

Figure 8.5 shows a single packet being processed. The functional objective is always to parse and process the raw data in a well-defined way. The data comes from some source object and is passed to some sink object.

A pointer to the raw data is passed upwards from the sink to the stencil by way of the data press. A pointer is used so that we avoid having to copy the data.¹⁴ When the stencil receives the raw data it takes ownership of it until it has finished processing it. At that point the ownership passes back to the data press as that class is responsible for routing the data. Normally the data would be passed on up the stack by calling `NewData()` on the sink.

To deal with the fact that packets can be fragmented and arrive in pieces, the stencil may not be able to fully parse the raw data passed to it when `NewData()` is called on it. However, when it has collected

¹³en.wikipedia.org/wiki/Endianess.

¹⁴Sometimes you can't avoid copying, for instance, when the raw data is stored in non-contiguous memory or you have to send it to another process over IPC.

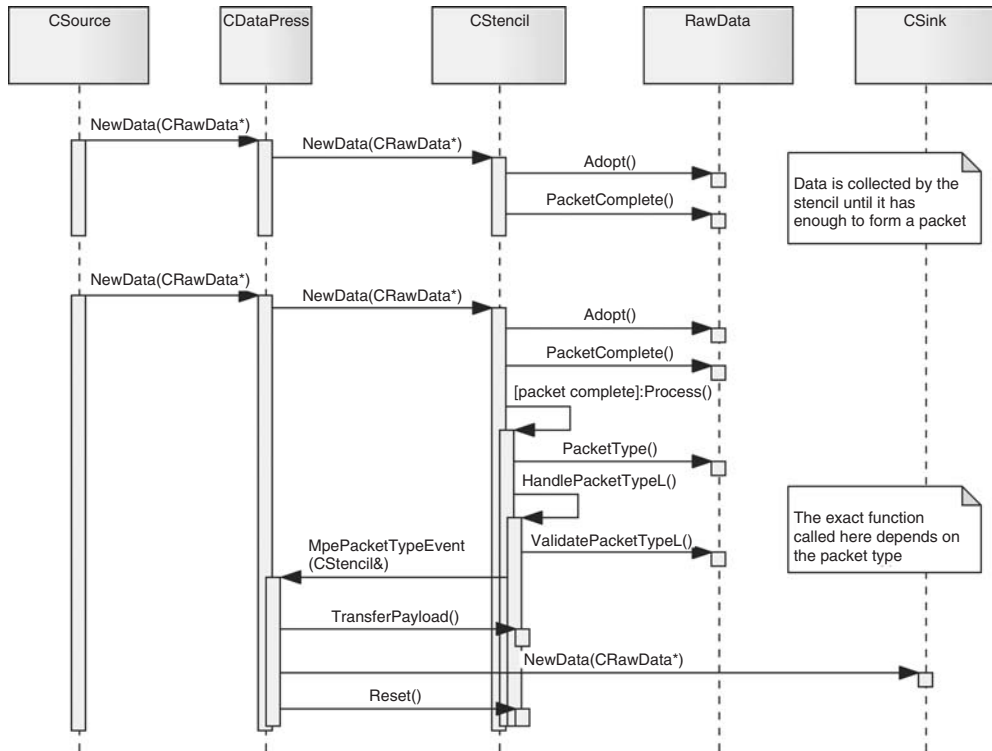


Figure 8.5 Dynamics of the *Data Press* pattern (valid packet)

enough data it begins its main task of parsing the packet in the following stages:

1. The packet is validated according to what type it is. At the very least, you should check whether the packet is of the correct size but you may need to perform more complex checks such as calculating checksums, etc.
2. The specific packet type needs to be processed. For some packet types this might be very simple but it can be much more complicated and, for instance, involve transforming the data by decompressing or decrypting it.
3. It informs the data press, through the `MpePacketEvents` event mixin, of the type of packet that has been processed.

Once the data press has been notified it should then deal with the event appropriately, which usually means driving its state machine and passing the payload of the packet to the sink. Having done this, the data press waits to be informed of the next piece of raw data to process.

The text above assumes that everything works normally. If only that were always the case! Unfortunately it's not and so you also need to handle the receiving of invalid or corrupt packets gracefully, as shown in Figure 8.6.

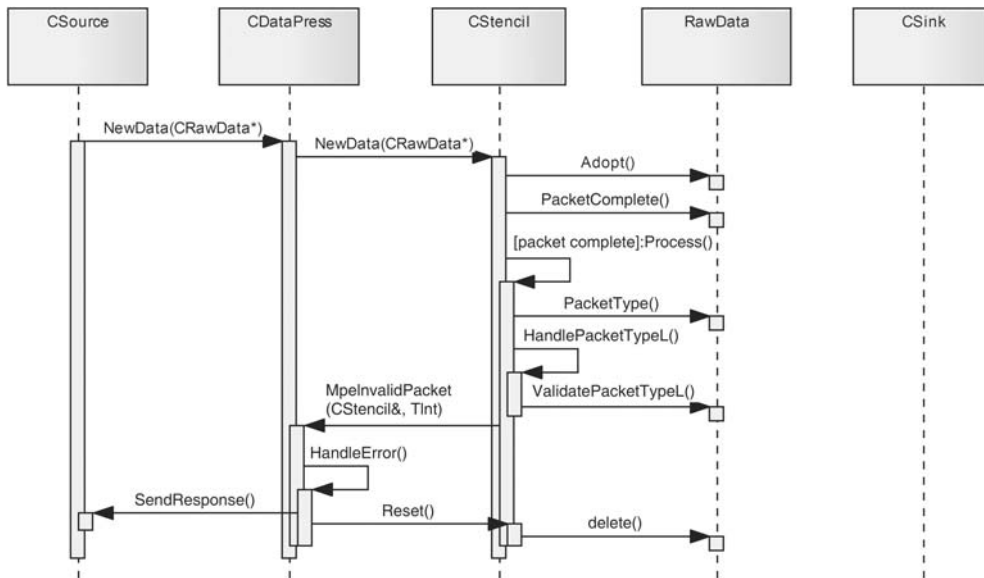


Figure 8.6 Dynamics of the *Data Press* pattern (invalid packet)

The exact error handling you need to perform depends on the domain to which you are applying this pattern. However, a common approach is to send a response back to the source that you identified a problem. This response might be a request to resend the data or it could be to disconnect the communication channel. Occasionally you also need to involve the sink in handling the error, although this isn't shown in Figure 8.6.

Implementation

Packet Events

Event Mixin (see page 93) is used to decouple the stencil and the data press and provide increased flexibility so that you can accommodate more easily any future changes. The interface described here is based on a warm event signal approach in which each of the methods are called simply to indicate that the stencil has identified a packet of a certain type. If the class implementing this interface wishes to obtain any further details then they can be accessed through the stencil. However, you might find

the hot event signal approach more suitable whereby the information that might be of interest to the derived class is passed as parameters to each method.¹⁵

```
class MPacketEvents
{
public:
    // This class has a function such as this for each possible packet type
    void MpePacketTypeEvent(CStencil& aPacket);
    void MpeInvalidPacket(CStencil& aPacket, TInt aError);
};
```

Data Press

This class needs to be implemented as a C class as it owns the stencil object. However, it would also be needed if this class contained data members used to store the state of your middleware component between individual packets.

Note that to be able to show the implementation of this pattern, we've chosen to use a fictional C class to represent the raw data type. However, this is only an example, and existing Symbian OS classes or other class types could be used instead.

```
class CDataPress : public CBase, public MPacketEvents
{
public:
    static CDataPress* NewL(CSource& aSource, CSink& aSink);
    ~CDataPress();
    void NewData(CRawData* aRawData);
public: // Functions from MPacketEvents
    void MpePacketTypeEvent(CStencil& aPacket);
    void MpeInvalidPacket(CStencil& aPacket, TInt aError);
private:
    CDataPress(CSource& aSource, CSink& aSink);
    void ConstructL();
    ...
private:
    CSource& iSource;
    CSink& iSink;
    CStencil* iStencil;

    // Further data members may be needed to track the state of the
    // component between packets
    ...
};
```

The following function is used to create the CStencil object and ensure it's accessible at all times. The other functions involved in the

¹⁵See Chapter 4 for more details on the difference between warm and hot event signals.

two-phase construction of this class aren't shown as they should be as per the standard approach.

```
void CDataPress::ConstructL()
{
    iStencil = CStencil::NewL();
}
```

The destructor of the class has to clean up the resources used by the class as appropriate. In particular it destroys the stencil class. This is shown to highlight the fact that the class is not deleted at any time between these two points and is re-used each time a new data packet is passed to the data press.

```
void CDataPress::~~CDataPress()
{
    delete iStencil;

    // You may need to de-register from either the sink or the source here
    ...
}
```

The function below is called by the source when it has processed the inbound data. This transfers ownership of the raw data to the data press class which itself immediately transfers ownership to the stencil.

```
void CDataPress::NewData(CRawData* aRawData)
{
    iStencil->NewData(aRawData);
}
```

The `MpePacketTypeEvent()` method shown below is just one of multiple similar functions called when the stencil has identified a particular packet type. This is where your domain-specific algorithm should be implemented. After this you should pass on the payload of the packet to the sink. Just the payload is passed on as the header of the packet is only relevant at the current layer. By calling `NewData()` in this way, the ownership of the raw data is passed to the sink.

```
void CDataPress::MpePacketTypeEvent(CStencil& aPacket)
{
    // Domain-specific implementation goes here. For instance, you might
    // need to change state as appropriate for the packet type.
    ...

    iSink->NewData(aPacket->TransferPayload());
    aPacket->Reset();
}
```

When an error occurs during the parsing of a packet, the following method is called. Exactly how you handle this is domain-specific but a common response is shown below:

```
void CDataPress::MpeInvalidPacket(CStencil& aPacket, TInt aError)
{
    iSource->SendResponse();
    aPacket.Reset();
}
```

Stencil

This class models the raw data as a packet, which is mostly about providing the data press with accessors to the fields of the packet header. To provide this information, it internally processes the raw data it is given to model depending on the packet type. The stencil is a classic C class because it owns data on the heap – the CRawData.

```
class CStencil : public CBase
{
public:
    static CStencil* NewL(MPacketEvents& aPacketObserver);
    ~CStencil();

    void NewData(CRawData* aRawData);
    CRawData* TransferPayload();

    // Accessors for the header fields
    TPacketType PacketType();
    THeaderField HeaderField(); // One per field in the header
private:
    CStencil(MPacketEvents& aPacketObserver);
    void ConstructL();
    void Reset();

    TBool PacketComplete();
    void ProcessL();
    void ValidatePacketTypeL(); // One per packet type
    void HandlePacketTypeL();  // One per packet type
private:
    enum TPacketType
    {
        EPacketType,
        EAnotherPacketType,
        ...
    };

    MPacketEvents& iPacketObserver;
    CRawData* iRawData;
};
```

The functions involved in the two-phase construction are as you might expect. However, the destructor needs to delete any raw data that it happens to be holding onto at the time:

```
void CStencil::~~CStencil()
{
    Reset();
}

void CStencil::Reset()
{
    delete iRawData;
    iRawData = NULL;
}
```

The real lifecycle of this class starts when it is provided with raw data to process:

```
void CStencil::NewData(CRawData* aRawData)
{
    // Take ownership of the data
    if (iRawData)
    {
        // Assumes CRawData has pre-allocated space to link in the
        // additional data without the possibility of a failure, e.g. by
        // having a TDbQueue data member
        iRawData.Append(aRawData);
    }
    else // We don't already have any data
    {
        iRawData = aRawData;
    }

    // Process the data once we have collected a complete packet
    if (PacketComplete())
    {
        TRAPD(err, ProcessL());
        if (err != KErrNone)
        {
            // There's been an error so notify the observer
            iPacketObserver->MpeInvalidPacket(*this, err);
        }
    }
    // else wait for more data
}
```

The above relies on the `ProcessL()` function to do the main work of validating and then interpreting the raw data appropriately. Note how this function uses *Escalate Errors* (see page 32) to simplify the error handling. The `NewData()` function acts as a high-level trap harness to catch any errors raised within this function so that they can be passed on to the data press to be dealt with.

However, all this function does directly is to ensure the correct parsing functions are called for each packet type. This is implemented below as a

simple `switch` statement but if performance is particularly critical then a lookup table can be more efficient, if less readable. You might find that arranging the order of the `switch` statement so that the most frequently occurring packet types come first provides a sufficient optimization that doesn't sacrifice the legibility of your code.

```
void CStencil::ProcessL()
{
    // Ensure that this function is only called when we've got raw data
    ASSERT(iRawData);

    switch (PacketType())
    {
    case EPacketType:
    {
        HandlePacketTypeL();
        break;
    }
    case EAnotherPacketType:
    {
        // Call the method to handle this packet type
        break;
    }
    // Deal with all the other packet types here
    default:
    {
        User::Leave(KErrCorrupt);
    }
    }
}
```

The handling of each packet type is also mostly domain-specific. It is in these functions that most of the work is done directly to process each individual packet type. First of all, the packet is validated. If this succeeds, it is safe to continue, which allows you to more easily parse the raw data by relying on it complying with the expected format.

This parsing might include transforming the raw data; if it can be done in place without the need for any additional RAM then all the better. However, if you can't transform it in place then you should ensure that, during the construction of the stencil, you pre-allocate sufficient RAM to perform any allocations that do occur. This is to prevent a system error from unexpectedly halting the parsing of a packet. Note that if you do transform the raw data you need to ensure that `iRawData` contains the final result as that is what is passed on to the sink.

```
void CStencil::HandlePacketTypeL() // One per packet type
{
    // Check the packet before doing anything else
    ValidatePacketTypeL();

    // Domain-specific handling
}
```

```
// Notify observer that this packet type has been identified
iPacketObserver->MpePacketTypeEvent(*this);
}
```

The private functions to validate each packet type are mostly domain-specific however there are some common themes.

```
void CStencil::ValidatePacketTypeL() // One per packet type
{
    // You almost always need to check the packet length
    if (PacketLength() < KPacketTypeMinLength ||
        PacketLength() > KPacketTypeMaxLength)
    {
        User::Leave(KErrCorrupt);
    }

    // You might also have a checksum field in the header that you can
    // match against the checksum on the whole packet that you've received
}
```

The following function is called by the data press to obtain the payload of the packet, usually to pass on to the sink. This function transfers the ownership of the payload data to the caller although the raw data forming the header is still owned by the stencil.

```
CRawData* CStencil::TransferPayload()
{
    TUint headerLength = HeaderLength();
    // Get raw data, point to the payload
    CRawData* ret = iRawData->Mid(headerLength);
    // Reduce amount of data owned to just the header
    iRawData->SetLength(headerLength);
    return ret;
}
```

The only functions that we have left to implement are the basic header field accessors such as `PacketType()`. These functions use the `CRawData` interface to extract the value required on demand. They have to be aware of issues such as managing the endianness of the data and lack of contiguous memory.

Consequences

Positives

- You reduce the error paths that have to be dealt with during the parsing by treating the stencil object as an *Immortal* (see page 53)

and allocating it upfront rather than allocating it just in time to handle each packet received.

- Latency is reduced as a consequence of using *Immortal* (see page 53) as the lifetime for the stencil, since this reduces the CPU cycles needed to parse each packet.
- RAM usage is reduced by sharing data and not fragmenting the heap with lots of small RAM allocations for the stencil object.
- Power usage is reduced as a natural consequence of the reduced RAM and CPU used.
- Maintenance costs are reduced. By encapsulating the processing of raw data into the stencil class, any changes to the format of the packet can be easily dealt with either by changing the implementation of the existing stencil class or by providing an alternative stencil type with the same overall interface but a different implementation. Maintenance costs are also kept down by having a clear separation between interpreting the raw data and the decisions based on it.

Negatives

- The stencil relies on the persistence of the raw data all the while it is processing it. This must be enforced outside the stencil class and potentially outside of your middleware component. For instance, it may be provided by convention between protocols in a data-processing stack. Similarly, the stencil class must be able to be sure that whilst it 'owns' the raw data it is working on, no-one else will try to manipulate the data at the same time.
- By pre-allocating the stencil, you reduce the overall RAM available in the system even if the stencil is not currently being used to model a packet. If the stencil just models the packet header this probably isn't significant. However, if RAM is required to perform transformations on the raw data it might be more of a problem.
- This pattern does not address directly how you store several instances of a message and keep state information about them. If you do need to do this then you'll need to extend this pattern by upgrading the data press class to store a collection of stencil classes. You'll probably find that the stencil classes are re-usable so long as you let them keep ownership of their raw data and pass a copy of it on the sink. If you need to store a lot of stencil objects then you should consider using Flyweight [Gamma *et al.*, 1994].

Example Resolved

The Symbian OS Bluetooth component AVDTP uses this pattern to implement the signaling channel described in the associated protocol specification. In this case, the data press is a class called `CSignallingChannel` which owns a single stencil, or `CAvdtpInboundSignallingMessage`, class. Rather than show you the entire class and duplicate much of the implementation shown above, we focus on the domain-specific functions that deal with the parsing, validation and handling of a selection of the AVDTP packet types.

The first example is based on the group of packets which deal with a device requesting a remote device to list the stream end points (SEPs) that it supports and hence what transport services it provides. These packets have the following specific types:

- **command** – when received from a remote device, this is a request to send a response detailing the locally supported SEPs
- **response accept** – received from a remote device in response from a discover command sent by this device and lists the SEPs the remote device supports
- **response reject** – as for response accept except that the initial request has been rejected for some reason (this is not shown below as it doesn't add much to the understanding of this pattern).

The data press object provides the following packet event functions for these packets:

```
class CSignallingChannel : public CBase, ... // Plus some other classes
{
public:
    void DiscoverIndication(CAvdtpInboundSignallingMessage& aMessage);
    void DiscoverConfirm(CAvdtpInboundSignallingMessage& aMessage);
    void InvalidPacket(CAvdtpInboundSignallingMessage &aMessage,
                      TInt aError);
    ...
};
```

Discover Command

The parsing for the discover command packet type is implemented as follows.

```
void CAvdtpInboundSignallingMessage::HandleDiscoverCommandL()
{
    // Simple validation
    CheckPacketLengthL(KAvdtpDiscoverCommandMinimumLength,
```

```

        KAvdtpDiscoverCommandMaximumLength);

// Simple handling
iSignallingChannel.DiscoverIndication(*this);
}

```

The following code shows how the data press handles this particular packet event without going into all the details. Note that `aMessage` isn't used here because this packet type has no payload.

```

void CSignallingChannel::DiscoverIndication(
    CAvdtpInboundSignallingMessage& aMessage)
{
    // Create transaction ready for the response
    CSignallingTransaction* transaction = PrepareSignallingResponse(
        aMessage->Id(), EResponseAccept, EAvdtpDiscover);
    if (transaction)
    {
        // Ask the session to contribute to the packet on behalf of
        // the client
        iClientSession->DiscoverIndication(*transaction);

        // We now deem the packet constructed and ready to send to the remote
        // device
        SendResponse(transaction);
    }

    aMessage.Reset(); // Clean up any remaining raw data
}

```

Discover Response Accept

The response accept packet provides a more complex parsing example. This is because the payload of the packet lists the SEPs the remote device supports and hence there is more data that needs to be validated.

```

void CAvdtpInboundSignallingMessage::HandleDiscoverResponseAcceptL()
{
    // More complex validation
    CheckPacketLengthL(KAvdtpDiscoverAcceptMinimumLength,
        KAvdtpDiscoverAcceptMaximumLength);

    // The remainder of the packet should be formed of a list of
    // information about the supported SEPs
    if (Payload().Length() % KAvdtpPacketSepInfoSize != 0)
    {
        User::Leave(KErrCorrupt);
    }
    // The packet must contain information about least one SEP
    if (Payload().Length() == 0)
    {

```

```

        User::Leave(KErrNotFound);
    }

    // Parsing complete
    iSignallingChannel.DiscoverConfirm(*this);
}

```

When the data press receives the discover confirm packet event, it passes it on up to the local client with the information about the remote device and, in this case, there is a payload that needs to be passed to the client. However, rather than just passing the data straight to the client, the data press uses the stencil to parse the list of SEPs so that it can store the information about what transaction services the remote device supports to help maintain the protocol's state.

```

void CSignallingChannel::DiscoverConfirm(
    CAvdtpInboundSignallingMessage& aMessage)
{
    CSignallingTransaction* transaction = FindTransaction(aMessage->Id());
    if (transaction)
    {
        TAvdtpSepList seps;
        aMessage->DiscoveredSeps(seps); // Used instead of TransferPayload()
        StoreDiscoveredSeps(seps);
        transaction->Client()->DiscoverConfirm(seps);
        delete transaction;
    }

    aMessage.Reset(); // Clean up any remaining raw data
}

```

The `DiscoveredSeps()` function above is a good example of an accessor function even though, strictly speaking, it isn't accessing the header information. Rather than passing the SEP list back as a return from the function, it takes a SEP list as an out parameter since this is more efficient for an object on the stack. Note that a data copy is needed here as the SEP list is going to be needed after the packet providing the information has been used up.

```

void CAvdtpInboundSignallingMessage::DiscoveredSeps(TAvdtpSepList& aSeps)
{
    // Parse the data out of the packet
    aSeps.iNumSeps = (Payload().Length()) / KAvdtpPacketSepInfoSize;

    for (TInt i=0; i<aSeps.iNumSeps; i++)
    {
        // Extract the info about each SEP
        // Note the use of accessor functions in the following code
        TAvdtpSepInfo sepInfo;
        sepInfo.SetSeid(SepSeid(i));
    }
}

```

```

    sepInfo.SetInUse(SepInUse(i));
    ...

    // Store the information
    aSeps.iSepsList.Append(sepInfo);
}
}

```

The function above relies on the following accessor functions that interpret the raw data directly. Note that in each of them a small amount of data needs to be copied. This is because the underlying raw data is stored in non-contiguous memory which can't be easily parsed. Since the size of the data is small, a copy is made to a contiguous memory location before the information in the required header field is extracted.

```

TSeid CAvdtpInboundSignallingMessage::SepSeid(TInt aSepOffset)
{
    TUint byte;
    iRawData.CopyOut(byte, (aSepOffset * KAvdtpPacketSepInfoSize), 1);
    TSeid seid = TSeid((byte >> KAvdtpPacketSeidOffset) &
                       KAvdtpPacketSeidMask);
    return seid;
}

TBool CAvdtpInboundSignallingMessage::SepInUse(TInt aSepOffset)
{
    TUint byte;
    iRawData.CopyOut(byte, (aSepOffset * KAvdtpPacketSepInfoSize), 1);
    return (byte & KAvdtpPacketSepInUseMask ? ETrue : EFalse);
}

```

Error Handling

When a parse error occurs, the protocol handles commands and responses differently since they differ in who is expecting a response:

- Commands have been received from a remote device and hence need to be sent an appropriate rejection response. The client, however, shouldn't be troubled by the error.
- Responses come from a remote device in response to a command the local device has sent and hence the client is expecting some kind of response. In this protocol, the remote device thinks it's done its job by responding and we don't automatically ask for a response to be sent again. That's left up to the client to decide.

This behavior is implemented in the following function called by the stencil after any point that it finds an error with the received packet. Note that it calls some accessors on the stencil which, if the packet, is invalid could cause problems. Hence you need to be very careful when doing

this to ensure that the function can never fail. This usually means the accessor has to return a value indicating that it couldn't determine the required information. For example, the `PacketType()` function returns `EInvalidPacketType` if this value can't be determined.

```
void CSignallingChannel::InvalidPacket(
                                CAvdtpInboundSignallingMessage& aMessage,
                                TInt aError)
{
    switch(aPacket.PacketType())
    {
    case EDiscoverCommand:
    {
        // Send reject response to the remote device
    }
    case EDiscoverResponseAccept:
    {
        // Pass the error up to the client
    }
    ...
    case EInvalidPacketType:
    {
        // Send the reject response to the remote device and
        // pass the error up to the client
    }
    }
}
```

Other Known Uses

- *Symbian's RfComm Protocol Implementation*
The pattern is used to process inbound messages for this Bluetooth serial port emulation protocol.
- *Symbian's TCP/IP stack*
The *Data Press* pattern is expressed as a template class that takes a variety of different stencils to process data for different protocols. This data press is not stored on the heap, but instead is created as needed on the stack. Since the data that is being processed is held in a separate heap that serves as a data bank, and is just referred to by a handle held by the template class (in a similar way to our example above), this is a cheap and efficient way of managing the data to be parsed.

Variants and Extensions

- *Long-Running Processing*
In the pattern described above, the processing of the raw data happens synchronously during the `NewData()` event and may result in `NewData()` being called on the source. As middleware components

are often stacked together, this could result in a long synchronous call from the bottom of the stack right up to the top. This would make the stack unresponsive to any new events during the processing of the packet by the whole stack. This is because the processing is likely to be running within the `RunL()` of a low-level active object and would hence be stalling the active scheduler. If this could be a problem then it is advisable during the `NewData()` event to setup an asynchronous callback to handle the actual processing of a packet, perhaps using the Symbian OS class `CAsyncCallback`.

- *Field Caching*
The implementation above shows the stencil header field accessors interpreting the raw data every time the function is called. For fields that are accessed frequently or are complex to parse, you might improve your execution time by caching their values as data members in the stencil.
- *Multiple Stencil Instances*
There can be scenarios where you are able to process multiple packets simultaneously. This can be done by the data press pre-allocating more than one instance of a stencil class. The main thing to remember when doing this is that each stencil must operate on a different piece of raw data.

References

- *Immortal* (see page 53) is used by the data press when pre-allocating a stencil object.
- *Escalate Errors* (see page 32) is used to simplify the error handling within the stencil object by escalating all errors up to the data press object to handle.
- *Event Mixin* (see page 93) is employed by the data process to provide the stencil with an interface through which it can pass the results of its interpretation of each packet.
- Flyweight [Gamma *et al.*, 1994] is related to this pattern. The idea of a Flyweight is to use a low footprint class to represent a small data feature, say a letter in a word-processing program. A small set of Flyweight objects are created and held for frequent reuse within a wider program. The objective is to reduce allocation costs but the Flyweight class is really more of a passive data model than the data press, which both models the data and modifies it.

9

Mapping Well-Known Patterns onto Symbian OS

In this chapter, we describe the following well-known design patterns and how they are applied and used on Symbian OS:

- *Model–View–Controller* (see page 332) was first described in [Buschmann *et al.*, 1996]. Symbian OS ensures every application uses this or a variant of this pattern so that they are easier to maintain, test and port between different mobile devices.
- *Singleton* (see page 346) was first described in [Gamma *et al.*, 1994]. This popular and sometimes controversial pattern (used to ensure an object is unique) is included because the standard mechanism of implementing it using writable static data hasn't always been available to Symbian OS developers. Whilst it is available in Symbian OS v9, alternative solutions are recommended for widely used DLLs due to the large impact it can have on memory usage.
- *Adapter* (see page 372) was first described in [Gamma *et al.*, 1994]. It is commonly used for at least the following three use cases: preserving compatibility by enabling an interface to continue to be provided that is re-implemented in terms of an upgraded interface; wrapping existing components to allow them to be ported to Symbian OS; and supporting a choice of components at run time by adapting them to a plug-in interface.
- *Handle–Body* (see page 385) was perhaps first described in [Coplien, 1991]. This pattern introduces an additional layer of abstraction between the client of an interface and its implementation. This is particularly useful for developers using Symbian OS as it not only ensures that your components are flexible enough to allow easy extension in future but also because it gives you more freedom to extend them in a compatible way without the clients of your component having to make any changes as a result.

Model–View–Controller

Intent Allow an interactive application to be easily extended, ported and tested by dividing the responsibility for managing, displaying and manipulating data between three cooperating classes.

AKA MVC

Problem

Context

You wish to create an interactive application with a flexible human–computer interface¹ that targets as many Symbian OS devices as possible across more than one GUI variant.

Summary

- You wish to be able to easily port your application between the different GUI variants based on Symbian OS.
- You want to be able to easily maintain and extend your application over time.
- You want to be able to test as much of your application as possible independently of the GUI variant to reduce your testing costs.

Description

Symbian OS supports multiple GUI variants, which are supplied by UI vendors as a layer on top of Symbian OS. By supporting multiple GUI variants, Symbian OS enables device creators to differentiate products based on form factor, UI ‘look and feel’, and interaction style. For application developers, the most important GUI variants are S60 and UIQ. The third GUI variant, MOAP, is widely used on Symbian OS phones in Japan, but is not open to native third-party applications so we don’t focus on it in this pattern.

Historically, S60 has been associated with classic smartphone designs based on small but high-resolution displays and key-driven operation, while UIQ has emphasized larger displays with pen-driven operation and flippable orientation, evolving from its early support for tablet-style devices to supporting the signature Sony Ericsson ‘flip’ phones. More

¹Derived from the intent of Model–View–Controller defined in [Buschmann *et al.*, 1996].

recently S60 and UIQ have shown signs of convergence, as UIQ3 has introduced support for purely key-driven interaction and S60 has indicated future intentions to support pen-based interaction styles.

Their 'look and feel', however, remain distinctively different, and there are significant differences in implementation detail. Each defines its own hierarchies of concrete controls (dialogs, windows, and so on), and while both share a task-based approach, there are subtle differences in their implementation of the task and application models.

Although the different GUI variants all derive from common Symbian OS frameworks, they encapsulate different vendor choices and assumptions about device form factors and interaction styles. This leads to non-trivial differences between the variants which means developers have to customize their applications for each GUI variant they wish to support. Any application that doesn't control its UI dependencies is more difficult to port between multiple, independently evolving target GUIs so we need to avoid a monolithic approach to creating applications.

On any operating system or platform, an interactive application with more than a trivial implementation typically has the following responsibilities, regardless of how they are assigned within the design:

- code that manages and manipulates the internal logical state of the application
- code that displays a representation of the current application state
- code that reacts to events from the end user to manipulate both of the above.

Example

Consider a contacts application which at its simplest allows the end user to store and later retrieve the contact details of the people that they know. Such an application needs to provide the following functionality:

- objects that are responsible for storing the contact information in the file system so that it's persisted across reboots of the device but is in an easy-to-access form
- objects that display the contact details to the end user so that they can interact with the data to add, update, search or delete contacts
- objects that listen for events generated by the end user such as the text they wish to search contacts for or the add contact menu item that has been selected.

In addition to these core requirements the application needs to work on S60 where the end user generally uses a joystick-style button to navigate

between contacts and UIQ where the user often taps the screen to select different contacts.

Solution

This pattern is based on the classic object-oriented abstraction of a graphically based, data-centric, interactive user application. The classic version of this pattern, as described in [Buschmann *et al.*, 1996], provides a ready-made template which is applicable to any GUI-based application on any platform. The classic pattern recognizes the three core elements of most interactive applications and recommends that each is designed as independent modules so that dependencies between the modules are minimized and the responsibilities are well encapsulated.

These modules are defined as follows:

- The *model* owns and manages the data that represents the internal state of the application and implements all application logic that directly changes that data, and hence the application state.
- The *view* provides a view for the end user onto the application state and ensures that it is always up to date with any changes to that state in the model.
- The *controller* provides the interaction logic and interfaces that enable the end user to interact with the application data, and hence manipulate the application state.

Structure

The structure of the classic version of the pattern is shown in Figure 9.1.

All GUI-based interactive applications on Symbian OS are implemented using a framework which is based on the classic version of this pattern, providing a similar level of abstraction even though it is different in some details:

- Symbian OS enforces clear separation between UI-dependent and UI-independent code.
- Symbian OS has historically emphasized the document-centric nature of applications although this has become less important and both S60 and UIQ minimize this aspect of the pattern.
- Symbian OS is based on a design in which application classes are derived from interfaces provided by the application frameworks to provide concrete implementations of framework-defined behaviors.²

²In fact, several frameworks are layered on top of each other, some contributed by Symbian and some by the UI vendor.

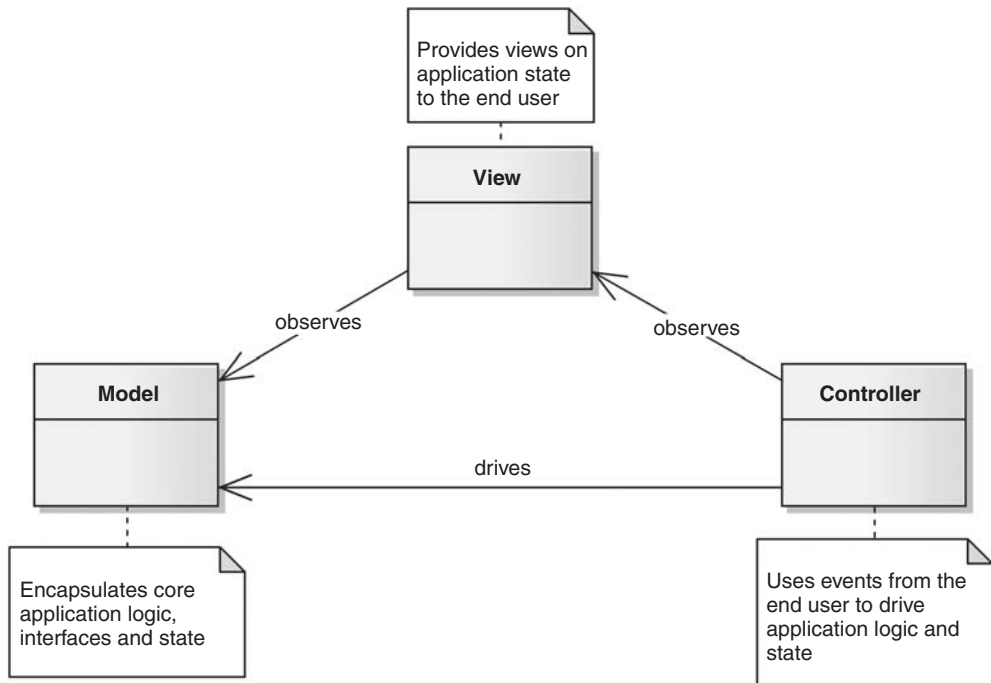


Figure 9.1 Classic structure of the *Model–View–Controller* pattern

As with any framework-based design, the application frameworks are intended to move complexity into system code. Thus the framework integrates individual applications and provides support for:

- the application lifecycle of installation, launch, moving to and from the background, and shutdown
- handling resources as well as associating documents with individual applications
- different display and graphics models including graphics contexts, concrete display classes, standard dialogs and other screen elements provided by the GUI variant
- a window hierarchy and system events.

This gives rise on Symbian OS to the basic structure shown in Figure 9.2.

Every Symbian OS application must implement the interfaces defined by the application frameworks. However, the application frameworks can be divided up into at least two layers: a generic layer provided by Symbian OS with a layer on top that provides the GUI variant's specializations. Individual applications then provide concrete implementations of the

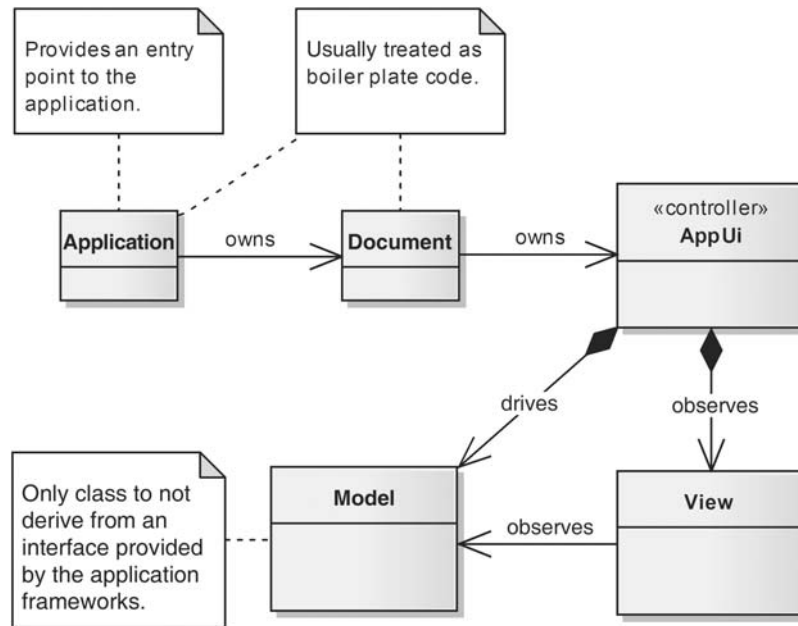


Figure 9.2 Structure of *Model–View–Controller* on Symbian OS

classes provided by the GUI variant and hence implement this basic pattern.

Originally, the Symbian OS use of this pattern assumed that applications are document-centric; an application is considered to be a document instance, either in a 'live' running state with an associated process or persisted as serialized data in a store somewhere in the system. However, subsequent usage has more typically been to make applications task-centric rather than document-centric so that the end user experience is typically organized around user tasks rather than applications and documents. End users are assumed to be more interested in tasks than in the applications that perform them and tasks are allowed to follow their natural course, which is often to flow across multiple applications; for example, starting with looking up a name in a Contacts application and moving seamlessly on to calling a number from the Phone application.

In practice, as in Figure 9.2, the document class is in many cases simply stubbed out. Instead, the `AppUi` directly owns a UI-independent application engine which encapsulates the application state and logic and is equivalent to the model in the classic version of this pattern.

Because all interaction between the model, the view and the controller is through framework-defined interfaces, there is a high degree of decoupling inherent in this structure that makes it relatively easy to swap out one or more classes for an alternative implementation.

Defining at least one view is essential for an application to display itself. While the classic Model-View-Controller pattern defines an explicit View class, the Symbian OS application frameworks do not; instead, a generic control class `CCoeControl` is defined, and in early releases of Symbian OS, application views were typically directly derived controls; in special cases³ they could be omitted altogether and replaced with a single concrete control or container. From Symbian OS v9.1 onwards, all standard application views derive from a suitable framework class supplied by the GUI variant. In UIQ3 in particular, the view classes provide important additional functionality, including support for view switching and view sharing between applications as well as integration with the new generic command structure.⁴

This structure allows the UI dependencies of the application to largely be confined to these views unlike the other two components, the controller and the model, which should have minimal dependences on the UI and hence are easily portable between different GUI variants. Views themselves are not intended to be portable but they are, to a high degree, decoupled from the model and hence the controller can be replaced for each version of your application specific to a GUI variant.

Dynamics

Figure 9.3 shows how the application classes interact.

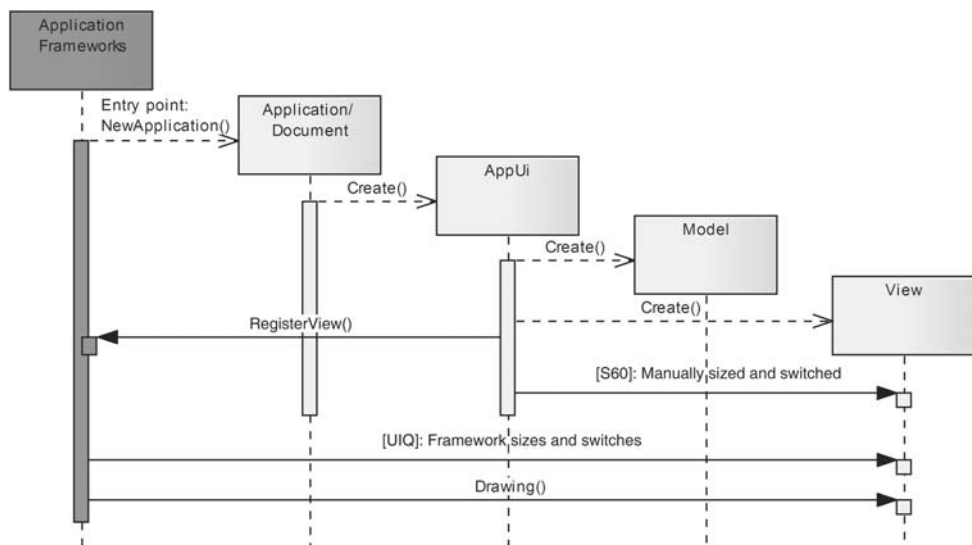


Figure 9.3 Dynamics of *Model-View-Controller* on Symbian OS

³For example, S60 2nd Edition.

⁴Introduced to support the multiple UI configurations of UIQ3.

Note that in S60 views are sized and switched by the `AppUi` whilst in UIQ this is done within the application frameworks. The view switching in particular highlights an important difference between the latest versions of UIQ and S60:

- In UIQ3, switching to *external views*, views that belong to a different application, is explicitly supported by the framework, encoded in the view persistence and 'backing out' behavior rules, and enshrined in application structure by the publication of external interface information in header files; the behavior of built-in UIQ applications depends on being able to drive one application into the view of a different application.
- In contrast, in the S60 view, persistence rules positively discourage leaving a view in the state of its last use or switching to an external view. Instead, S60 takes the approach of embedding one application instance within another to achieve a similar effect. S60 applications rarely have their state affected by another application, whereas in UIQ this is the norm.

An important point to remember is that, just as the classic version of this pattern relies on Observer [Gamma *et al.*, 1994] to allow the controller to react to events generated by the view and the view to reflect the application state stored in the model, the Symbian OS version of this pattern also relies on an event-driven programming approach to keep the three main classes aligned whilst allowing them to remain largely decoupled. If you introduce your own model class, you should ensure that you build on this approach to avoid introducing hard dependencies that will reduce your ability to port your application in future.

Implementation

Overview

There is no such thing as a generic Symbian OS application, because at a detailed level application design depends on specific features of GUI variants. However, at a fundamental level the underlying Symbian OS frameworks impose a common blueprint on all applications irrespective of UI variant:

- The GUI variant framework implementations derive from the underlying Symbian OS frameworks and therefore share a common structure across all devices.
- While specific details of GUI behavior differ between UI variants, the significant differences are encapsulated in those variants and come 'for free' with the use of the specific application frameworks for the associated devices.

- The main differences between the UI variants are the details of resource file usage and application lifecycle implementation details. This is in addition to more obvious differences to the application's own UI that reflect the 'look and feel' of the UI variant such as view layout and navigation, menu layout and behavior and the layout of application-independent screen areas, which includes the general status and task bars that are always shown by a device.
- The independence of Symbian OS from the GUI variant running on top of it encourages the partitioning of application code between GUI-independent application engine code and GUI-dependent display and user interaction code.

The Symbian OS application frameworks are hidden from concrete applications by the application frameworks of the GUI variant for the device you are currently targeting. Beneath the GUI variant, Symbian OS defines the following levels of UI and application frameworks:

- *Uikon* is the most abstract framework. It defines the generic application framework, free of any 'look and feel' policies, from which GUI variants derive a concrete framework implementation. It includes base classes derived from, as well as concrete classes used by, the derived application frameworks.
- *Application Architecture*⁵ provides a less abstract implementation of *Uikon* interfaces that define the UI-independent application lifecycle and responsibilities, including the relationships between applications, data, and resources.
- *Control Environment*⁶ provides a less abstract implementation of *Uikon* interfaces to supply base classes and an environment for abstract UI *controls*. These controls are window-using, possibly nested, rectangular screen areas that accept user input and system events. In particular this framework provides abstract interfaces for graphics contexts, the windowing model, and event dispatching.

All Symbian OS applications derive their basic application classes from GUI variant base classes, including (from v9) application view base classes. Table 9.1 shows a mapping between the base Symbian OS framework classes, the GUI variant base classes used by applications and the classic MVC model.

As Table 9.1 shows, there is no immediate equivalent for the Symbian OS Application class in the classic MVC model and MVC views can have a number of alternative implementations in Symbian OS applications.

⁵Also known as AppArc.

⁶Also known as CONE.

Table 9.1 *Model–View–Controller Inheritance Hierarchy*

S60 Base Class	UIQ Base Class	Symbian OS Base Classes and Associated Framework	Classic MVC Equivalent
CAknApplication	CQikApplication	CEikApplication (Uikon) extends CApaApplication (AppArc)	N/A
CAknDocument	CQikDocument	CEikDocument (Uikon) extends CApaDocument (AppArc)	Model
CAknViewAppUi and, for non-view-based applications, CAknAppUi	CQikAppUi	CEikAppUi (Uikon) extends CCoeAppUi (CONE)	Controller
CAknView and, for non-view-based applications, CCoeControl NB: All S60 view base classes are control-owning, i.e. they own an instance of a CCoeControl.	CQikViewBase (for ‘single page’ views), CQikMulti-PageViewBase (for ‘multi-page’ views), and CQikViewDialog (for simple ‘dialog’ views). NB: All UIQ view base classes are CCoeControl derived i.e. they are controls.	CCoeControl (CONE)	View

The following code fragments demonstrate the degree to which a Symbian OS application follows a generic structure, irrespective of the GUI variant it targets. The variations between S60 and UIQ certainly cannot be ignored and in some cases are quite subtle. They can, however, be quite easily identified and isolated.

Application

The top-level application class binds the application into main application frameworks and hence into the application lifecycle.

In the code for an Application class, the only difference between S60 and UIQ is the base class derived from. In either case, you still need to implement the entry point that the framework uses to launch

the application as well as a method to return the application UID. This class is also responsible for the next object in the chain, the application Document, whether or not it is seriously used.

```
class CMyApplication : public // CAknApplication OR CQikApplication
{
public:
    // Return Application's UID
    TUid AppDllUid() const;
protected:
    // Create the document
    CApaDocument* CreateDocumentL();
private:
    // Called by the framework to create a new application instance
    CApaApplication* NewApplication();
};
```

Document

This class was intended to encapsulate the application data and core logic however it is usually not seriously used though it still needs to be provided to allow the application to interact with the application frameworks. The class is responsible for creating the next object in the chain, the AppUi. If it is used in earnest, such as by a file-based application, the document reads its initial state from a file. Again the only difference between S60 and UIQ is the base class derived from.

```
class CMyDocument : public // CAknDocument OR CQikDocument
{
public:
    // Constructors / destructor omitted

    // Create the App UI
    CEikAppUi* CreateAppUiL();
};
```

AppUi

The AppUi provides the application-specific handling of events given to it by the application frameworks as a result of system events such as those generated by the end user. It is generally responsible for creating the application's model and drives it as a result of these events.

In addition, the AppUi provides the UI classes and defines the commands which are called from the menu choices command switch handler. It does this by creating the main default view and registers it with the framework although it is not expected to destroy the view.

The responsibilities of the `AppUi` class differ between S60 and UIQ. In S60, the `AppUi` is responsible for command handling, which follows the original Symbian OS pattern; in UIQ3, a new command-handling framework moves responsibility for command-handling to view classes, enabling commands to become view-specific.

```
class CMyAppUi : public // CAknAppUi OR CQikAppUi
{
public:
    // The application frameworks requires the default C++ constructor to
    // be public
    CMyAppUi();
    ...
private:
    // The AppUi owns the main application view
    CMyAppView* iAppView;
private:
    // S60 apps only, cf. new UIQ command-handling frameworks
    void HandleCommandL(TInt aCommand);
    void HandleStatusPaneSizeChange();
};
```

View

S60 and UIQ diverge most significantly in the way that they handle views, reflecting important, but subtle, differences in the underlying treatment of views and, in particular, of view switching. In UIQ, Direct Navigation Links⁷ enable applications to drive views belonging to other applications; in S60, on the other hand, applications typically collaborate using embedding. Both view derivation and view responsibilities and behavior are different between these GUI variants.

The following is a conventional S60 view class, which is derived from `CCoeControl`; more recent S60 view classes differ in that they are not control derived, but do own controls:

```
class CMyS60View : public CCoeControl
{
public:
    // Constructors/destructor omitted

    // Handle drawing and size changes
    void Draw(const TRect& aRect) const;
    virtual void SizeChanged();
};
```

The UIQ application view derives from one of the new UIQ view base classes which are themselves derived from `CCoeControl`:

⁷See UIQ 3 SDK » UIQ Developer Library » UIQ Style Guide » 5 Navigation.

```
class CUIQTtemplateView : public CQikViewBase
{
public:
    // Constructors/destructor omitted

    // Return a globally unique view id that can be used to DNL to this
    // view
    TVwsViewId ViewId() const;

    // Commands are handled in the context of a view
    void HandleCommandL(CQikCommand& aCommand);
};
```

Consequences

In practice, almost any application written for Symbian OS that has a UI is obliged to use this pattern. The possible exceptions are special case dialog-based applications that require only a very simple dialog-like interface, which both S60 and UIQ support; for example, settings management and other 'single function' applications. Even in these cases however, the application must implement a simplified version of the standard application pattern.⁸ This means that there is no real question of a trade-off for applications between using it and not using it. It is, however, useful to understand the influence it has on your application.

Positives

- Development costs are reduced as the strong framework support allows applications to follow a standard template with minimal custom code to behave as a fully integrated, well-formed application.
- Maintenance costs are reduced because of both the reuse of the applications frameworks proven in many releases of Symbian OS and the use of a well-known design pattern.⁹
- Porting of applications between GUI variants is made easier through the isolation of UI dependencies from core application functionality which makes it easier to reuse the same core application logic across multiple GUI variants based on Symbian OS.
- Testing costs are reduced as the high degree of decoupling between the individual MVC components allows each to be easily isolated and unit tested.
- Clear and structured application designs are enforced.

⁸There are a handful of exceptions such as console or server applications which do not have to use this pattern.

⁹The classic version of this pattern was possibly the first pattern to be described in 'pattern' terms.

Negatives

- For developers new to Symbian OS, especially those unfamiliar with object-oriented and framework programming, there is an inevitable overhead in learning how to implement applications using this pattern. Especially as this pattern differs from the classic implementation.

In practice, this isn't a significant issue as there are many sources of information and help for application developers that include the SDK for each GUI variant in addition to the Symbian Developer Library. IDEs, such as Carbide,¹⁰ often provide wizards to create complete template applications and, last but not least, there are many good examples (including complete source code) and tutorials both online and in books from Symbian Press such as [Babin, 2007] and [Harrison and Shackman, 2007].

Example Resolved

For a full set of example code on how a contacts application can be created, please see:

- <Nokia S60 3rd Edition SDK Installation Directory >\S60Ex\Contacts
- UIQ 3 SDK » UIQ Developer Library » UIQ Examples » QLayout – this describes an example application that has a view similar to that of a contacts application.

Other Known Uses

All Symbian OS applications, by definition, follow the Symbian variant of this pattern. Some well-known applications are:

- the phone application, which allows you to make phone calls
- the agenda application, which provides a calendar, allows you to create a todo list, and so on
- the messaging application, which allows you to send SMS messages, emails, and so on.

Variants and Extensions

- *Document–View*

Document–View is a common variant which Windows developers will be familiar with through the Microsoft Foundation classes (MFC).¹¹ In the Document–View pattern, the Controller and View

¹⁰ www.forum.nokia.com/main/resources/tools_and_sdks/carbide.

¹¹ [msdn.microsoft.com/en-us/library/4x1xy43a\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/4x1xy43a(VS.80).aspx).

are combined into a View. This may be done in cases where the UI dependencies are extensive and cannot be decoupled from the event handling of the controller.

Java developers are likely also to have encountered Document–View, although as [Cooper, 2000, pp. 3–4] notes, the original Java Swing implementation in version 1.2 uses the full MVC pattern for its components (e.g. `JTable` and `JList`).

References

- This pattern builds on Observer [Gamma *et al.*, 1994] to allow the three main classes to stay aligned whilst allowing them to remain decoupled.
- For more information on the standard industry view of MVC, see en.wikipedia.org/wiki/Model-view-controller.
- For a general background on GUI architectures, see martinfowler.com/eaDev/uiArchs.html.
- The Sony Ericsson developer website provides guidance on porting S60 applications to UIQ at developer.sonyericsson.com/site/global/techsupport/tipstrickscode/symbian/p_automatic_conversion_of_series_60_projects.jsp.
- The Forum Nokia website provides guidance on porting UIQ applications to S60 at www.forum.nokia.com/search/?keywords=porting+uiq.

Singleton

Intent Ensure a class has only one instance and provide a global point of access to it.

AKA Highlander¹²

Problem

Context

You wish to 'ensure a class only has one instance and provide a global point of access to it.' [Gamma *et al.*, 1994]

Summary

- A single instance of a class is required by a component or even the whole system. For example, to provide a single point of access to a physical device, such as a camera, or to a resource or resource pool, such as a logging object, a thread or a block of memory.¹³
- A way of synchronizing access to the single instance may be needed if you wish it to be in scope for different parts of a system and used by several threads or processes.
- Instantiation of the object should be controllable. Instantiation should either be deferred until the first time it is needed, or the object should be instantiated in advance of other objects, to ensure a particular order of initialization.

Description

This pattern is one of the simplest design patterns and arguably the most popular pattern found in [Gamma *et al.*, 1994]. Hence in this discussion, rather than examine the pattern in itself, we place emphasis on how to implement it on Symbian OS, covering a range of different circumstances. For detailed discussion of the pattern itself covering, for example, the overall consequences of the pattern or how to subclass a singleton class, please see [Gamma *et al.*, 1994].

The pattern involves just one class that provides a global point of access to a single instance, which instantiates itself, either using *Immortal* (see page 53) or *Lazy Allocation* (see page 63). However, there is a basic

¹²Thanks to Mal Minhas and Mark Jacobs for this alternative name, which is inspired by the movie of the same name, the tag of which is 'There can be only one'.

¹³See Pooled Allocation [Weir and Noble, 2000].

problem for developers working on Symbian OS, which is its incomplete support for writable static data in DLLs. To explain the issue, we must digress briefly: first of all, to discuss what we mean by ‘incomplete support for writable static data’ and then to explain why it’s a big deal.

The Writable Static Data Restriction on Symbian OS

For the purposes of this discussion, writable static data (WSD) is any modifiable globally-scoped variable declared outside of a function and any function-scoped static variable.

WSD was not allowed in DLLs built to run on target devices based on Symbian OS v8.1a or earlier.¹⁴ For an explanation of how this limitation came about, see [Willee, Nov 2007] and [Stichbury, 2004]. WSD has always been allowed in EXEs but, prior to Symbian OS v9, this was of little help to application developers because applications were built as DLLs.

With the advent of Symbian OS v9, the situation changed for the better. The use of WSD in target DLLs is now possible. However, because it can be expensive in terms of memory usage, it is not recommended when writing a shared DLL that is intended to be loaded into a number of processes. This is because WSD in a DLL typically consumes 4 KB of RAM for each process which loads the DLL.

When a process loads its first DLL containing WSD, it creates a single chunk to store the WSD. The minimum size of a chunk is 4 KB, since this is the smallest possible page size, so at least that amount of memory is consumed, irrespective of how much static data is actually required. Any memory not used for WSD is wasted.¹⁵

Since the memory is per process, the potential memory wastage is:¹⁶

$$(4 \text{ KB} - \text{WSD bytes}) \times \text{number of client processes}$$

If, for example, a DLL is used by four processes, that’s potentially an ‘invisible’ cost of almost 16 KB since the memory occupied by the WSD itself is usually in the order of bytes.

Furthermore, DLLs that use WSD are not fully supported by the Symbian OS emulator that runs on Windows. Because of the way the emulator is implemented on top of Windows, it can only load a DLL with WSD into a single emulated process. If a second emulated process attempts to load the same DLL on the emulator, it will fail with `KErrNotSupported`.¹⁷

¹⁴See Table 9.2 for the corresponding versions of S60 and UIQ.

¹⁵However, if subsequent DLLs are loaded into the same process and also use WSD, the same chunk is used, rather than a separate chunk (at least another 4 KB) for every DLL.

¹⁶Assuming you don’t use more than 4 KB of WSD.

¹⁷Symbian OS v9.4 introduces a workaround (see [Willee, Nov 2007] for a more advanced discussion).

There are a number of occasions where the benefits of using WSD in a DLL outweigh the disadvantages; for example, when porting code that uses WSD significantly and for DLLs that will only ever be loaded into one process. As a third-party developer, you may find the memory costs of WSD acceptable if you create a DLL that is intended only to be loaded into a single application. However, if you are working in device creation, perhaps creating a shared DLL that ships within Symbian OS, one of the UI platforms, or on a phone handset, the trade-offs are different. Your DLL may be used by a number of processes and the memory costs and constraints are such that using WSD is less likely to be justifiable.

Application developers are no longer affected as severely by the limitation, because a change in the application framework architecture in Symbian OS v9 means that all applications are now EXEs rather than DLLs. Writable static data has always been allowed in EXEs built for target devices, so an application developer can use WSD if it is required. Table 9.2 summarizes the support for WSD in DLLs and applications on various versions of the Symbian UI platforms.

Table 9.2 WSD Support Across Versions of Symbian OS and GUI Variants

UI Platform	Symbian OS version	WSD allowed in DLLs?	WSD allowed in applications?
UIQ 2.x	v7.0	No	No
S60 1st and 2nd Editions	v6.1, v7.0s, v8.0a, v8.1a		
UIQ 3	v9.x	Yes, but not recommended due to the limitations on the Windows emulator and potentially large memory consumption.	Yes, since applications are EXEs.
S60 3rd Edition			

Why Is the Limitation on WSD a Problem when Implementing Singleton?

Figure 9.4 shows the UML diagram for a typical *Singleton*.

The classic implementation of the *Singleton* pattern in C++ uses WSD, as shown below:

```
class Singleton
{
public:
    static Singleton* Instance();
    // Operations supplied by Singleton
}
```

```

private:
    Singleton(); // Implementation omitted for clarity
    ~Singleton(); // Implementation omitted for clarity
private:
    static Singleton* pInstance_; // WSD
};

/*static*/ Singleton* Singleton::pInstance_ = NULL;
/*static*/ Singleton* Singleton::Instance()
{
    if (!pInstance_)
        pInstance_ = new Singleton;

    return (pInstance_);
}

```

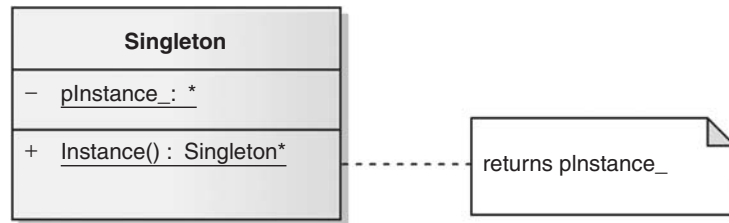


Figure 9.4 UML diagram for a typical *Singleton*

Because of the constraint on the use of WSD, the classic implementation of *Singleton* cannot be used in DLLs before Symbian OS v9.¹⁸ An alternative implementation, which we'll discuss shortly, must be used instead. This alternative is also recommended when writing DLLs for newer versions of Symbian OS that do allow WSD in DLLs, simply to avoid unnecessary waste of memory and to enable complete testing on the Windows emulator.

However, let's have another brief digression and discuss the classic code above. As you can see, the singleton object effectively owns and creates itself. The code required to instantiate *Singleton* is private, so client code cannot create the singleton instance directly. Only the static function `Instance()`, which is a member of the class, can create `pInstance_` and it does so the first time that it is called. This approach guarantees the singularity of the object, in effect, enforcing it at compile time, and also delays the construction until the object is needed, which

¹⁸Rather confusingly, DLL code that used WSD did actually build and run on the Windows emulator on versions of Symbian OS that didn't support the use of WSD in DLLs. It was only when the same code was compiled for target hardware that the use of WSD was actually flagged as an error – a time-consuming irritant for those working primarily on the emulator to build and debug in the early stages of a project. You can find out how to track down WSD if you're not using it deliberately in FAQ 0329 on the Symbian Developer Network (developer.symbian.com/main/support/faqs).

is known as *Lazy Allocation* (see page 63). Using that pattern can be advantageous if an object is 'expensive' to instantiate and may not be used. A good example is a logging class that is only used if an error occurs when the code is running. There is no point in initializing and holding open a resource such as a file server session if it is not needed when the code logic runs normally.

Hazards of Singleton

This pattern is often embraced for its simplicity by those new to design patterns, but it presents a number of potential pitfalls to the unwary programmer, especially the cleanup and thread-safety of the singleton class itself.

For example, we've just discussed the construction of `Singleton`, but what about the other end of its lifetime? Note that the destructor is private¹⁹ to prevent callers of `Instance()` from deleting `pInstance_` accidentally or on purpose. However, this does raise some interesting questions: When should the singleton instance destroy itself? How do we de-allocate memory and release resource handles? [Alexandrescu, 2001] is a good place to find out more and we'll discuss it in the Solution section.

Also, how do we ensure that the implementation is thread-safe if the singleton instance needs to be accessed by more than one thread? After all, it's intended to be used as a shared global resource, so it's quite possible that multiple threads could simultaneously attempt to create or access the singleton. The classical implementation we examined earlier is not thread-safe, because race conditions are possible when separate threads attempt to create the singleton. At worst, a memory leak arises if the first request for the singleton by one thread is interrupted by the thread scheduler and another thread then runs and also makes a request for the singleton object.

Here again is the code for `Instance()`, with line numbers to aid the discussion:

```

1 /*static*/ Singleton* Singleton::Instance()
2 {
3     if (!pInstance_)
4     {
5         pInstance_ = new Singleton;
6     }
7     return (pInstance_);
8 }
```

Consider the following scenario: Thread A runs and calls `Instance()`. The singleton `pInstance_` has not yet been created, so the code runs on to line 5, but it is then interrupted by the thread scheduler before

¹⁹If the `Singleton` class is ever intended to be sub-classed, it should be made protected.

`pInstance_` can be created. Thread B starts executing, also calls `Instance()` and receives the same result as thread A when it tests the `pInstance_` pointer. Thread B runs to line 5, creates the singleton instance and continues successfully. However, when Thread A resumes, it runs from line 5, and also instantiates a `Singleton` object, assigning it to `pInstance_`. Suddenly there are two `Singleton` objects when there should be just one, causing a memory leak because the one `pInstance_` pointer references only the latter object.

When *Singleton* is used with multiple threads, race conditions clearly need to be avoided and we'll discuss how to do this in the Solution section below. [Myers and Alexandrescu, 2004] provides an even more detailed analysis.

Should You Use Singleton?

Be wary of just reaching for this pattern as there are a number of subtleties that can catch you out later. This pattern increases the coupling of your classes by making the singleton accessible 'everywhere', as well as making unit testing more difficult as a singleton carries its state with it as long as your program lasts. There are a number of articles that discuss this in more depth but here are just a few:

- Do we really need singletons? [Saumont, 2007]
- Use your singletons wisely [Rainsberger, 2001]
- Why singletons are evil [Densmore, 2004]

Synopsis

We've covered quite a lot of ground already, so let's summarize the problems that we need to address in the solution:

- Symbian-specific issues
 - The classic implementation of *Singleton* does not allow for potential out-of-memory conditions or two-phase construction. What is the preferred approach in Symbian C++?
 - *Singleton* cannot be implemented using the classic solution, which depends on WSD, where WSD is not supported (in DLLs in Symbian OS v8.1a or earlier) or is not recommended (DLLs in Symbian OS v9). How can you work around this?
- General issues
 - When should a singleton instance destroy itself?
 - How should an implementation of *Singleton* cope with access by multiple threads?

Example

On Symbian OS, the window server (WSERV) provides a low-level API to the system's user interface devices – screens, keyboard and pointer. The API is extensive and complex and would make most application development unnecessarily convoluted if used directly. For example, to receive events caused by the end user interacting with a list box, you would need to use a window server session, as well as *Active Objects* (see page 133), to handle asynchronous events raised by WSERV in response to input from the end user.

The control framework (CONE) hides much of this complexity and provides a simplified API that meets the requirements of most applications. CONE encapsulates the use of *Active Objects* (see page 133) and simplifies the interaction with the Window Server by providing a single, high-level class, `CCoeEnv`, for use by application programmers. `CCoeEnv` also provides simplified access to drawing functions, fonts, and resource files which are used by many applications. The class is also responsible for creation of the cleanup stack for the application, for opening sessions to the window server, file server, screen device, application's window group and a graphics context.

Only one instance of `CCoeEnv` is needed per thread since it provides thread-specific information. However, `CCoeEnv` is intended for use by a potentially large number of different classes within an application, so the instance of `CCoeEnv` should be globally accessible within each thread.

Solution

Depending on the scope across which you require the singleton to be unique there are a number of different ways of working around the problem as shown in Table 9.3.

Some of the implementations discussed are accompanied by their own issues. For example, at the time of writing, Implementation C should be regarded as illustrative rather than recommended on Symbian OS.

Implementation A: Classic Singleton in a Single Thread

Constraints on the Singleton

- Must only be used within a single thread.
- Can be used within an EXE on any version of Symbian OS or in a DLL on Symbian OS v9.x if absolutely necessary.
- WSD may be used.

Table 9.3 Synopsis of Solutions

Implementation	Maximum Required Scope for the Singleton	Details	Requires WSD?
A	Within a GUI application or other EXEs such as console applications or servers	Implement normally with only minor modification for Symbian C++.	Yes
B	Within a DLL for a specific thread	Use thread local storage (TLS).	No
C	Across multiple threads	Use a thread synchronization mechanism to prevent race conditions.	Yes
D	Across multiple threads	Manage initialization and access to the TLS data as each secondary thread starts running.	No
E	Across multiple processes	Use the Symbian OS client–server framework.	No

Details

On Symbian OS, the implementation of *Singleton* must take into account the possibility of instantiation failure. For example, a leave because there is insufficient memory to allocate the singleton instance. One possible technique is to implement a standard `NewL()` factory function that will be familiar to Symbian C++ developers:

```
class CSingleton : public CBase
{
public:
    // To access the singleton instance
    static CSingleton& InstanceL();
private: // The implementations of the following are omitted for clarity
    CSingleton();
    ~CSingleton();
    static CSingleton* NewL();
    void ConstructL();
private:
    static CSingleton* iSingletonInstance; // WSD
};
```

```
/*static*/ CSingleton& CSingleton::InstanceL()
{
    if (!iSingletonInstance)
    {
        iSingletonInstance = CSingleton::NewL();
    }
    return (*iSingletonInstance);
}
```

Inside `InstanceL()`, the `NewL()` factory method is called to create the instance. The main issue is what to do if this fails? One option would be to ignore any errors and pass back a pointer to the singleton that might be `NULL`. However, this puts the burden on clients to check for it being `NULL`. They are likely to forget at some point and get a `KERN-EXEC 3` panic. Alternatively, we could panic directly if we fail to allocate the singleton which at least tells us specifically what went wrong but still takes the whole thread down which is unlikely to be the desired outcome. The only reasonable approach is to use *Escalate Errors* (see page 32) and leave (see Figure 9.5). This means the client would have to deliberately trap the function and willfully ignore the error to get a panic. While this is entirely possible, it won't happen by accident. It also gives the client the opportunity to try a less severe response to resolving the error than terminating the thread.

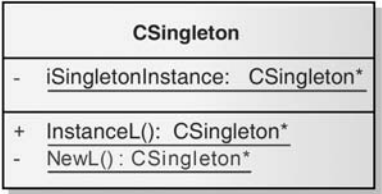


Figure 9.5 Structure of Implementation A (basic)

Note: On most versions of Symbian OS, the tool chain disables the use of WSD in DLLs by default. To enable WSD you must add the `EPOCALLOWDLLDATA` keyword to your MMP file. However, there are some builds of Symbian OS (for example, Symbian OS v9.3 in S60 3rd Edition FP2) that allow WSD in DLLs without the need to use the `EPOCALLOWDLLDATA` keyword. In addition, the currently supported version of the GCC-E compiler has a defect such that DLLs with static data *may* cause a panic during loading. The issue, and how to work around it, is discussed further in [Willee, Nov 2007].

A variant of this approach is to give clients of the singleton more control and allow them to instantiate the singleton instance separately from actually using it (see Figure 9.6), allowing them to choose when to handle any possible errors that occur.

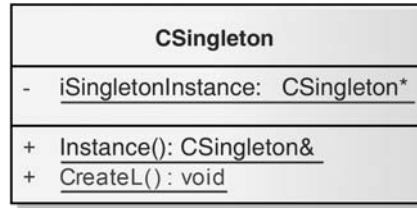


Figure 9.6 Structure of Implementation A (lifetime managed)

A separate method can then be supplied to guarantee access to the singleton instance once it has been instantiated.

```
class CSingleton : public CBase
{
public:
    // To create the singleton instance
    static void CreateL();
    // To access the singleton instance
    static CSingleton& Instance();
private: // The implementations of the following are omitted for clarity
    static CSingleton* NewL();
    CSingleton();
    ~CSingleton();
    void ConstructL();
private:
    static CSingleton* iSingletonInstance; // WSD
};

/*static*/ void CSingleton::CreateL()
{
    // Flags up multiple calls in debug builds
    ASSERT(!iSingletonInstance);

    // Create the singleton if it doesn't already exist
    if (!iSingletonInstance)
    {
        iSingletonInstance = CSingleton::NewL();
    }
}

/*static*/ CSingleton& CSingleton::Instance()
{
    ASSERT(iSingletonInstance); // Fail Fast (see page 17)
    return (*iSingletonInstance);
}
```

This approach gives the caller more flexibility. Only one call to a method that can fail is necessary, in order to instantiate the singleton instance. Upon instantiation, the singleton is guaranteed to be returned as a reference, so removing the requirement for pointer verification or leave-safe code.

Positive Consequences

- Simple to understand and implement.
- Little impact on performance.

Negative Consequences

- Uses 4 KB per process for processes that don't already load a DLL using WSD.
- Can only be used safely by one thread.

Implementation B: TLS Singleton in a Single Thread

Constraints on the Singleton

- Must only be used within a single thread.
- May be used within an EXE or a DLL on any version of Symbian OS.
- WSD may not be used.

Thread local storage (TLS) can be used to implement *Singleton* where WSD is not supported or recommended. This implementation can be used in DLLs on all versions of Symbian OS, and in EXEs if desired. It is not thread-safe.

Details

TLS is a single storage area per thread, one machine word in size (32 bits on Symbian OS v9).²⁰ A pointer to the singleton object is saved to the TLS area and, whenever the data is required, the pointer in TLS is used to access it.

The operations for accessing TLS are found in class `D11`, which is defined in `e32std.h`:

```
class D11
{
public:
    static TInt SetTls(TAny* aPtr);
    static TAny* Tls();
    static void FreeTls();
    ...
};
```

Adapting Implementation A to use TLS does not change the API of `CSingleton`, except that the methods to create and access the singleton, `CreateL()` and `Instance()`, must be exported in order to

²⁰Note that, since TLS is provided on a per-thread basis, this implementation does not share the data between threads. See Implementations C, D or E for multi-threaded singletons.

be accessible to client code outside the DLL, as must any specific methods that the singleton class provides. The implementation of `CreateL()` and `Instance()` must be modified as follows:

```
EXPORT_C /*static*/ void CSingleton::CreateL()
{
    ASSERT(!Dll::Tls());
    if (!Dll::Tls())
    { // No singleton instance exists yet so create one
        CSingleton* singleton = new(LEAVE) CSingleton();
        CleanupStack::PushL(singleton);
        singleton->ConstructL();
        User::LeaveIfError(Dll::SetTls(singleton));
        CleanupStack::Pop(singleton);
    }
}

EXPORT_C /*static*/ CSingleton& CSingleton::Instance()
{
    CSingleton* singleton = static_cast<CSingleton*>(Dll::Tls());
    ASSERT(singleton);
    return (*singleton);
}
```

Positive Consequences

- Relatively straightforward to understand and implement.
- Circumvents the WSD limitations in Symbian OS DLLs.

Negative Consequences

- The price of using TLS instead of direct access to WSD is performance. Data may be retrieved from TLS more slowly than direct access²¹ through a RAM lookup. On ARMv6 CPU architectures, it is about 30 times slower however this should be considerably improved by subsequent CPU architectures.
- Maintainability is reduced because of the need to manage the single TLS slot per thread. If TLS is used for anything else in the thread, all the TLS data must be put into a single class and accessed as appropriate through the TLS pointer. This can be difficult to maintain.

Implementation C: Classic Thread-safe Singleton Within a Process

Constraints on the Singleton

- Must be used within a single process but may be used across multiple threads.

²¹See [Sales, 2005].

- May be used within an EXE on any version of Symbian OS or in a DLL on Symbian OS v9.x, if absolutely necessary.
- WSD may be used.

Details

We earlier demonstrated that the `Instance()` method of the classical implementation of *Singleton* is not thread-safe – race conditions may occur that result in a memory leak by creating two instances of the class.

At first sight, there appears to be a trivial solution to the problem of making *Singleton* leave-safe – add a mutex. However, consider the following pseudocode:

```
/*static*/ Singleton& Singleton::Instance()
{
    Mutex.Lock(); // Pseudocode
    if (!pInstance_)
    {
        pInstance_ = new Singleton();
    }
    Mutex.Unlock(); // Pseudocode
    return (*pInstance_);
}
```

The use of `Mutex` has prevented the potential for race conditions and ensures that only one thread executes the test for the existence of `pInstance_` at any one time. However, it also means that the mutex must be locked and unlocked every time the singleton is accessed even though the singleton creation race condition we described earlier can only occur once. The acquisition of the mutex lock, and its release, results in an overhead that is normally unnecessary. The solution may have looked straightforward, but the overhead is far from desirable.

To work around this, perhaps we could only lock the mutex after the test for the existence of `pInstance_` ? This gives us the following pseudocode:

```
/*static*/ Singleton& Singleton::Instance()
1 {
2   if (!pInstance_)
3   {
4     Mutex.Lock(); // Pseudocode
5     pInstance_ = new Singleton();
6     Mutex.Unlock(); // Pseudocode
7   }
8   return (*pInstance_);
9 }
```

However, this has reintroduced the possibility of a race-condition and thus a memory leak. Assume thread A runs and tests `pInstance_`

in line 2. If it is `NULL`, the code must lock the mutex and create the singleton. However, suppose thread A is interrupted, prior to executing line 4, by thread B. The singleton hasn't been created yet, so thread B passes through the `if` statement and on to line 4, locking the mutex and creating `pInstance_`, before unlocking the mutex in line 6. When thread A runs again, it is from line 4, and it proceeds to lock the mutex and create a second instance of the singleton. The check for the existence of `pInstance_` comes too early if it occurs only before the lock.

This gives rise to an implementation that double checks the `pInstance_` pointer: that is, checking it after acquiring the lock *as well as* before. This is known as the Double-Checked Locking Pattern (DCLP) and was first outlined in [Schmidt and Harrison, 1996]:

```

/*static*/ Singleton& Singleton::Instance()
1 {
2   if (!pInstance_)
3   {
4     Mutex.Lock(); // Pseudocode
5     if (!pInstance_)
6     {
7       pInstance_ = new Singleton();
8     }
9     Mutex.Unlock(); // Pseudocode
10  }
11  return (*pInstance_);
12 }

```

Now the mutex is only acquired if `pInstance_` is not yet initialized, which reduces the run-time overhead to the minimum, but the potential for a race condition is eliminated, because a check is also performed after the mutex is locked. It is these two checks that give rise to the name of DCLP.

Now let's consider the simplest DCLP implementation in Symbian C++, using WSD.²² As we saw in Implementation A, there are two possible approaches:

- Provide a method to provide access to the singleton instance and create it if it does not already exist.
- Split the creation of the singleton instance from the method required to access it, to make it easier for calling code to use the singleton without having to handle errors when memory resources are low.

²²Please note that the implementation shown is unsafe, as described in the consequences sections. The DCLP should be used carefully until later versions of Symbian OS provide a memory barrier to guarantee the order of execution on both single- and multiple-processor hardware. Even then, there will still be the problem of using a globally-named mutex, which is insecure as we highlight in the consequences sections.

The `InstanceL()` method for the former approach is as follows:²³

```
/*static*/ CSingleton& CSingleton::InstanceL()
{
    if (!iSingletonInstance) // Check to see if the singleton exists
    {
        RMutex mutex;
        // Open a global named RMutex (or RFastLock)
        User::LeaveIfError(mutex.OpenGlobal(KSingletonMutex));
        mutex.Wait(); // Lock the mutex

        if (!iSingletonInstance) // Perform the second check
        {
            iSingletonInstance = CSingleton::NewL();
        }

        mutex.Signal(); // Unlock the mutex
    }

    return (*iSingletonInstance);
}
```

A very basic example of how the singleton may be used is shown below. Three threads, a main thread and two secondary threads, access the singleton, and call a method called `DoSomething()` to illustrate its use:

```
TInt Thread1EntryPoint(TAny* /*aParameters*/)
{
    TRAPD(err, CSingleton::InstanceL().DoSomething());
    ...
    return err;
}

TInt Thread2EntryPoint(TAny* /*aParameters*/)
{
    TRAPD(err, CSingleton::InstanceL().DoSomething());
    ...
    return err;
}

// Main (parent) thread code
// Creates a named global mutex
RMutex mutex;
User::LeaveIfError(mutex.CreateGlobal(KSingletonMutex));
CleanupClosePushL(mutex);
...
RThread thread1;
User::LeaveIfError(thread1.Create(_L("Thread1"), Thread1EntryPoint,
                                KDefaultStackSize, KMinHeapSize,
                                KMaxHeapSize, NULL));
```

²³The two-phase construction code is as shown in Implementation A.


```

CleanupClosePushL(thread1);

RThread thread2;
User::LeaveIfError(thread2.Create(_L("Thread2"), Thread2EntryPoint,
                                KDefaultStackSize, KMinHeapSize,
                                KMaxHeapSize, NULL));

CleanupClosePushL(thread2);

// Start the threads off 'at the same time'
thread1.Resume();
thread2.Resume();

TRAPD(err, CSingleton::InstanceL().DoSomething());

// Not shown here: Don't forget to clean up the threads, the mutex and
// the singleton
...

```

The alternative approach described above splits creation of the singleton from access to it, to make it easier for calling code to use. With the addition of DCLP, the code for this approach looks as follows:

```

*static*/ CSingleton& CSingleton::Instance()
{
    ASSERT(iSingletonInstance);
    return (*iSingletonInstance);
}

/*static*/ void CSingleton::CreateL()
{
    if (!iSingletonInstance) // Check to see if the singleton exists
    {
        RMutex mutex;
        User::LeaveIfError(mutex.OpenGlobal(KSingletonMutex));
        mutex.Wait(); // Lock the mutex

        if (!iSingletonInstance) // Second check
        {
            iSingletonInstance = CSingleton::NewL();
        }

        mutex.Signal(); // Unlock the mutex
    }
}

```

The code that uses the singleton is now more straightforward, since the `Instance()` method always returns a valid instance and no errors have to be handled. However, to be sure that the singleton has been created, each thread must call the `CSingleton::CreateL()` method once, unless the thread is guaranteed to start *after* one or more other threads have already called `CSingleton::CreateL()`.

Positive Consequences

- Prevents race conditions and provides a leave-safe solution for use where WSD is available and multiple threads must be able to access a singleton.
- Use of double-checked locking optimizes performance by locking the mutex on singleton creation only and not on every access.

Negative Consequences

- 4 KB of RAM is used for the WSD in the one process sharing the singleton.
- There is additional complexity of implementation and use compared to Implementation A. If it's possible to ensure access to the singleton by a single thread only, Implementation A is preferable.
- Safe use of DCLP cannot currently be guaranteed. There is no way to prevent a compiler from reordering the code in such a way that the singleton pointer becomes globally visible before all the singleton is fully constructed. In effect, one thread could access a singleton that is only partially constructed. This is a particular hazard on multiprocessor systems featuring a 'relaxed' memory model to commit writes to main memory in bursts in order of address, rather than sequentially in the order they occur.²⁴ Symmetric multiprocessing is likely to be found in future versions of Symbian OS. However, it is also a problem on single processor systems. The 'volatile' keyword can potentially be used to write a safe DCLP implementation but careful examination of the compiler documentation and a detailed understanding of the ISO C++ specification is required. Even then the resulting code is compiler-dependent and non-portable.
- The example given is insecure because it uses a globally accessible mutex. Global objects are inherently prone to security and robustness issues since a thread running in any process in the system can open them by name and, in this case, block legitimate threads attempting to create the singleton by calling `Wait()` without ever calling `Signal()`.
- You cannot share a singleton across multiple threads executing in different processes. This is because each process has its own address

²⁴[Alexandrescu, 2001] suggests that you should perform triple-checked locking by first checking your compiler documentation before going on to implement the DCLP! If you plan to use the DCLP, you should certainly consult [Myers and Alexandrescu, 2004] for a detailed discussion of the potential hazards. The final section of that paper suggests alternative ways to add thread-safety to *Singleton* that avoid the use of the DCLP.

space and a thread running in one process cannot access the address space of another process to get at the singleton object.

Implementation D: Thread-Managed Singleton within a Process

Constraints on the Singleton

- Must be used within a single process but may be used across multiple threads.
- May be used within an EXE or a DLL on any version of Symbian OS.
- WSD may not be used.

Details

As its name suggests, the storage word used by TLS is local to a thread and each thread in a process has its *own* storage location. Where TLS is used to access a singleton in a multi-threaded environment, the pointer that references the location of the singleton must be passed to *each* thread. This is not a problem since all threads are in the same process and they can all access the memory address. This can be done using the appropriate parameter of `RThread::Create()`. If this is not done, when the new thread calls `Dll::Tls()`, it receives a NULL pointer.

This implementation does not need to use DCLP because, for the threads to share the singleton, its instantiation must be controlled and the location of an existing singleton passed to threads as they are created. The main thread creates the singleton before the other threads exist, thus the code is single-threaded at the point of instantiation and does not need to be thread-safe. Once the singleton is available, its location may be passed to other threads, which can then access it. Thus this implementation is described as 'thread managed' because the singleton cannot be lazily allocated, but must be managed by the parent thread.

Let's look at some code to clarify how this works. Firstly, the `CSingleton` class exports two additional methods that are used by calling code to retrieve the singleton instance pointer from the main thread, `SingletonPtr()`, and set it in the created thread, `InitSingleton()`. These methods are added so that code in a process that loads and uses the singleton-providing DLL can initialize and use the singleton without needing to be aware of how the singleton class is implemented.

```
class CSingleton : public CBase
{
public:
    // To create the singleton instance
    IMPORT_C static void CreateL();
    // To access the singleton instance
```

```

    IMPORT_C static CSingleton& Instance();
    // To get the location of the singleton so that it can be passed to a
    // new thread
    IMPORT_C static TAny* SingletonPtr();
    // To initialize access to the singleton in a new thread
    IMPORT_C static TInt InitSingleton(TAny* aLocation);
private:
    CSingleton();
    ~CSingleton();
    void ConstructL();
};

EXPORT_C /*static*/ CSingleton& CSingleton::Instance()
{
    CSingleton* singleton = static_cast<CSingleton*>(Dll::Tls());
    return (*singleton);
}

EXPORT_C /*static*/ void CSingleton::CreateL()
{
    {
        ASSERT(!Dll::Tls());
        if (!Dll::Tls()) // No singleton instance exists yet so create one
        {
            CSingleton* singleton = new(ELeave) CSingleton();
            CleanupStack::PushL(singleton);
            singleton->ConstructL();
            User::LeaveIfError(Dll::SetTls(singleton));
            CleanupStack::Pop(singleton);
        }
    }
}

EXPORT_C TAny* CSingleton::SingletonPtr()
{
    {
        return (Dll::Tls());
    }
}

EXPORT_C TInt CSingleton::InitSingleton(TAny* aLocation)
{
    {
        return (Dll::SetTls(aLocation));
    }
}

```

The code for the main thread of a process creates the singleton instance and, as an illustration, creates a secondary thread, in much the same way as we've seen in previous snippets:

```

// Main (parent) thread must create the singleton
CSingleton::CreateL();

//Create a secondary thread
RThread thread1;
User::LeaveIfError(thread1.Create(_L("Thread1"), Thread1EntryPoint,
                                KDefaultStackSize, KMinHeapSize,
                                KMaxHeapSize,
                                CSingleton::SingletonPtr()));

CleanupClosePushL(thread1);

// Resume thread1, etc...

```

Note that the thread creation function now takes the return value of `CSingleton::SingletonPtr()` as a parameter value (the value is the return value of `Dll::Tls()` – giving access to the TLS data of the main thread). The parameter value must then be passed to `CSingleton::InitSingleton()` in the secondary thread:

```
TInt Thread1EntryPoint(TAny* aParam)
{
    TInt err = CSingleton::InitSingleton(aParam);
    if (err == KErrNone )
    { // This thread can now access the singleton in the parent thread
        ...
    }

    return err;
}
```

Positive Consequences

- *Singleton* can be implemented in a DLL and accessed by multiple threads in the process into which the DLL is loaded.

Negative Consequences

- Accidental complexity of implementation and use. For example, code that executes in a secondary thread must initialize its own TLS data by explicitly acquiring and setting the location of the singleton in the parent thread.
- The main thread must control the creation of every secondary thread.
- This solution works for multiple threads executing in a single process, but cannot be used to share a singleton across multiple threads executing in different processes. This is because each process has its own address space and a thread running in one process cannot access the address space of another process.

Implementation E: System-wide Singleton

Constraints on the Singleton

- Can be used anywhere in the system across multiple processes or threads.
- May be used within an EXE or a DLL on any version of Symbian OS.
- WSD may not be used.

Details

As described in the consequences sections of Implementations C and D, those implementations allow a singleton to be accessed by multiple threads within a single Symbian OS process, but not by multiple processes. The singleton implementation is only a single instance within a process. If several processes load a DLL which provides Implementation D, each process would have its own copy of a singleton to preserve an important quality of processes – that their memory is entirely isolated from any other process.

Where a system-wide singleton is required on Symbian OS, one process must be assigned ownership of the singleton object itself. Each time another process, or client, wishes to use the singleton it has to make an IPC request to get a copy of the singleton. To enable this, the provider of the singleton should make a client-side DLL available that exposes the singleton interface to these other processes but is implemented to use IPC transparently to obtain the copy of the actual singleton.

Having hidden the IPC details from clients behind what is effectively a Proxy [Gamma *et al.*, 1994] based on the singleton interface, you have a choice of which IPC mechanism to use to transmit the copy of the singleton data. Here are some mechanisms provided by Symbian OS that you can use to perform thread-safe IPC without the need for WSD or the DCLP:

- *Client–Server* (see page 182)
This has been the most commonly used mechanism to implement this pattern as it has existed in Symbian OS for many versions now. The server owns the singleton and sends copies on request to its clients.
- *Publish and Subscribe* kernel service
You may find that if all you wish to do is provide a system-wide singleton then using the Publish and Subscribe kernel service is easier to implement. Note that whilst this is the same technology as used in *Publish and Subscribe* (see page 114), the way it is used differs. For instance, whilst the owning process does indeed ‘publish’ the singleton, the client-side DLL does not ‘subscribe’ for notification of changes. Instead, it simply calls `RProperty::Get()` when an instance of the singleton is needed.
- *Message Queues*
Mentioned for completeness here as they could be used but the other alternatives are either more familiar or simpler to use.

The disadvantage of having a system-wide singleton is the impact on run-time performance. The main overhead arises from the context switch involved in the IPC message to and from the process owning the singleton and, if the singleton contains a lot of data, the copying of the singleton.

Positive Consequences

- Allows access by multiple processes and thus provides a single, system-wide, instance.
- The Symbian OS IPC mechanisms manage the multiple requests for access to a single shared resource and handle most of the problems of making the singleton thread-safe.
- Calling code can be unaware of whether it is accessing a singleton – the client-side implementation is simple to use, while the process supplying the singleton can be customized as necessary.

Negative Consequences

- Additional execution time caused by the IPC and copying of the singleton data although any implementation of Singleton that supports access from multiple processes will also be affected by this.
- The implementation of the singleton is necessarily more complex, requiring creation of a client-side DLL and use of an IPC mechanism. The implementation can be heavyweight if only a few properties or operations are offered by the singleton.

Solution Summary

Figure 9.7 summarizes the decision tree for selecting a singleton implementation on Symbian OS.

Example Resolved

The control environment (CONE) implements class `CCoeEnv` as a singleton class. Implementation B is used and a pointer to the singleton is stored in TLS. Internally, `CCoeEnv` accesses the singleton instance through a set of inline functions, as follows:

```
// coetls.h
class CCoeEnv;

inline CCoeEnv* TheCoe() { return((CCoeEnv*)Dll::Tls()); }
inline TInt SetTheCoe(CCoeEnv* aCoe) { return (Dll::SetTls(aCoe)); }
inline void FreeTheCoe() { Dll::FreeTls(); }
```

The constructor of `CCoeEnv` uses an assertion statement to confirm that the TLS slot returned by `TheCoe()` is uninitialized. It then sets the TLS word to point to itself, by calling `SetTheCoe(this)`.

When an object of any class, instantiated within a thread that has a control environment (i.e. the main thread of an application) needs access

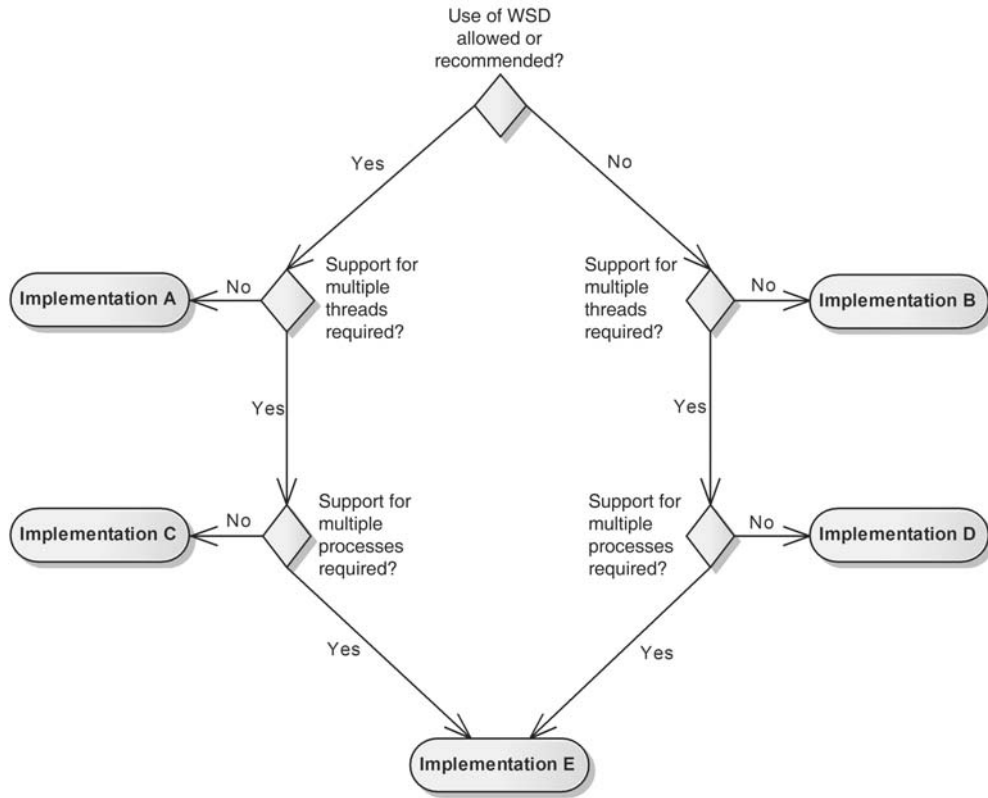


Figure 9.7 Symbian OS *Singleton* implementation decision tree

to the `CCoeEnv` singleton, it may simply include the `coemain.h` header and call `CCoeEnv::Static()` which is implemented as follows in `coemain.cpp`:

```
EXPORT_C CCoeEnv* CCoeEnv::Static()
{
    return(TheCoe());
}
```

It can then be used by applications as follows:

```
CCoeEnv::Static()->CreateResourceReaderLC(aReader, aResourceId);
```

Alternatively, `CCoeControl::ControlEnv()` can be used by applications to retrieve the singleton instance. However, this method returns a cached pointer to the `CCoeEnv` singleton, which the `CCoeControl` class stores in its constructor (by making a call to `TheCoe()`).

The use of a cached pointer is more efficient than calling `CCoeEnv::Static()` because it avoids making an executive call to `Dll:Tls()` to access the singleton. Within the application framework, using a cached pointer is acceptable, because the lifetime of the `CCoeEnv` singleton is controlled and, if the application is running, the `CCoeEnv` object is guaranteed to exist (and to be at the same location as when the `CCoeControl` constructor first cached it).

In general, it is not considered a good thing to cache pointers to singleton objects for future access, because it makes the design inflexible. For example, the singleton can never be deleted without providing some kind of mechanism to ensure that those using a cached pointer don't access dead memory. The `Instance()` method should generally be used to access the singleton, because that allows more flexibility to the provider of the singleton. For example, it could save on memory by deleting the object if available memory drops below a certain threshold or if it has not been accessed recently or regularly.

The implementations given in the Solution section anticipate that some clients may mistakenly cache a pointer to the singleton 'because they can' and instead return a reference from `Instance()`, which is more difficult to code around (although it is still possible).

Other Known Uses

This pattern is widely used across Symbian OS since virtually every application makes use of the `CCoeEnv` singleton in its thread whilst every server is itself a singleton for the whole system.

Variants and Extensions

- *Destroying the Singleton*

All the implementations discussed have specified the destructor for the singleton class as private. This is because, if the destructor was public, it is possible that the singleton instance could be deleted, leaving the singleton class to hand out 'dangling references' to the deleted instance. Unless the design is such that this can be guaranteed not to occur, it is preferable to prevent it.

Since the singleton class has responsibility for creation of the instance, it must generally have responsibility of ownership and, ultimately, cleanup.

One option could be to allow the singleton destruction to be implicit. C++ deletes static objects automatically at program termination and the language guarantees that an object's destructor will be called and space reclaimed at that time. It doesn't guarantee the calling order, but if there is only one singleton in the system, or the order of destruction is unimportant, this is an option when Implementations

A or C (using WSD) are chosen. Please see [Vlissides, 1996] for more information about this approach, which requires the use of a friend class to destroy the singleton.

Another option is to use the `atexit` function provided by the standard C library, to register a cleanup function to be called explicitly when the process terminates. The cleanup function can be a member of the singleton class and simply delete the singleton instance. Please see [Alexandrescu, 2001] for further details.

However, you may want to destroy the singleton before the process terminates (for example, to free up memory if the object is no longer needed). In this case, you have to consider a mechanism such as reference counting to avoid dangling references arising from premature deletion. For example, to extend Implementation A, with the addition of a static `Close()` method:

```
class CSingleton : public CBase
{
public:
    // To create the singleton instance
    static void CreateL();
    // To access the singleton instance
    static CSingleton& Instance();
    static void Close();
private: // The implementations of the following are omitted for
        clarity
    CSingleton* NewL();
    CSingleton();
    ~CSingleton();
    void ConstructL();
private:
    static CSingleton* iSingletonInstance;
    TInt iRefCount;
};

/*static*/ void CSingleton::CreateL()
{
    ++(iSingletonInstance->iRefCount);
    if (!iSingletonInstance)
    { // Create the singleton if it doesn't already exist
        iSingletonInstance = CSingleton::NewL();
    }
}

/*static*/ CSingleton& CSingleton::Instance()
{
    ASSERT(iSingletonInstance);
    return (*iSingletonInstance);
}
```

```
// Must be called once for each call to CreateL()
/*static*/ void CSingleton::Close()
{
    if (--(iSingletonInstance->iRefCount)<=0)
    {
        delete iSingletonInstance;
        iSingletonInstance = NULL;
    }
}
```

References

- *Lazy Allocation* (see page 63) can be used when allocating the *Singleton* instance.
- *Client–Server* (see page 182) may be used for Implementation E.
- [Saumont, 2007], [Rainsberger, 2001] and [Densmore, 2004] discuss the problems of using this pattern.
- In Monostate, as described in [Ball and Crawford, 1997], there may be more than one object of a class but all share the same state.

Adapter

Intent Transform calls on one interface into calls on another, incompatible, interface without changing existing components.

AKA Emulator [Buschmann *et al.*, 1996]

Problem

Context

There is a pre-existing component, the adaptee, which exposes an interface that you wish to use via an alternative incompatible interface, the target.

Summary

You need to resolve one or more of the following:

- You wish to reduce your development costs by porting a pre-existing component, the adaptee, to a new environment.
- You have re-engineered a component, the adaptee, but you wish to support the old interface, the target, to preserve compatibility for legacy clients.
- You wish to increase the functionality and flexibility of a component by re-using existing components, the adaptees, at run time.

Description

This pattern was originally described in [Gamma *et al.*, 1994]. Hence in this discussion rather than examine the pattern itself in too much detail, we place emphasis on how it can be used on Symbian OS, covering a range of diverse circumstances.

The context of this pattern of having to match incompatible interfaces can arise in numerous different situations. Software based on Symbian OS is subject to many pressures all of which can give rise to problems of mismatching interfaces: it is an evolving system with a large user base, which means that maintaining interface compatibility from release to release is important not just for the operating system itself but also for any services and frameworks built on top of it. It targets a domain which covers a wide and diverse set of technologies, many of which are standards-driven and must interoperate, and it is a specialized operating system, carefully evolved to work in a demanding environment.

In addition, you may not be able to change existing components to make compatible what were incompatible interfaces. The reasons for this range from pragmatic ones of engineering cost or risk, to business reasons (for example, licensing terms), to quality reasons (avoiding tight coupling between logically independent components).

Each of the following kinds of problem demonstrates the same common factor: the need, for whatever reason, to make incompatible interfaces work together, without changing existing components:

- *Preserving compatibility*
When the legacy functionality provided by one component is made obsolete by changes that, as your system evolves, introduce new, alternative functionality which meet at least the same requirements, clients relying on the legacy behavior must still be supported, if only for a transitional period. For reasons of maintainability and code size, it may not be feasible to leave the old component in place alongside the newer version.
- *Wrapping components to ease porting*
For existing components created for a different environment from the one you would like to use it in, it may be impossible or undesirable to make changes to the component. This commonly arises when integrating an industry-standard implementation of a specific technology.
The interfaces presented to a component in the Symbian OS environment may be incompatible with what the component expects and, vice versa, the interfaces it presents may be incompatible with what potential Symbian OS clients of the component expect.
For Symbian OS, the problem is complicated by the fact that native C++ interfaces are often not directly usable by ported components written in standard C or C++; and standard C or C++ client interfaces of ported components are typically not directly usable by native Symbian OS clients.
- *Run-time adaptation*
Different components that all store state may present very different settings interfaces, but updating their settings may be the responsibility of a single agent or service. Hardwiring knowledge of each different settings interface into the updating service may be a poor design choice, because it breaks modularity and flexibility by forcing a tight coupling between many different components, possibly in quite different parts of the system.

Example

Here we give an example for each of the three main uses of this pattern identified in the Description above.

Preserving Compatibility

Symbian OS provides a communications database which is a persistent store of the data required to configure communications components or to make connections with remote servers. Until Symbian OS v9.1, the interface to this communications database was via the `CCommsDatabase` provided by the `CommDB` component that used the existing OS database service `DBMS`.

However, in Symbian OS v9.1, Symbian introduced a new component called `CommsDat` that provides the same interface as the old `CommDB`, `CCommsDatabase`, but uses the Symbian OS Central Repository instead. This move makes use of the extra functionality provided by the Central Repository compared to `DBMS`, such as standard tools to set up and configure the Central Repository during development as well as performance enhancements such as caching repositories used by multiple clients.

Since the underlying data is stored in a different place, the old `CommDB` component cannot simply be maintained alongside the new `CommsDat` component. Hence an alternative solution is needed which maintains compatibility to allow the legacy clients of `CommDB` to continue to work but which uses the new storage location.

Wrapping Components to Ease Porting

Before Symbian OS v9.2, a native database component known as `DBMS` provided all clients with a database service supporting the SQL specification. However, for v9.2, Symbian decided to port the open source SQLite component²⁵ to Symbian OS. One of the main reasons for this was that there was a requirement to provide more SQL features, such as triggers, that aren't provided by `DBMS`.

One decision taken early on was not to attempt to preserve compatibility with the old `DBMS` interface but instead to provide an interface to SQLite alongside it. This was because, whilst SQLite provided more SQL functionality than `DBMS`, it didn't, at the time, provide a superset, i.e. there is some SQL functionality that `DBMS` supported that SQLite didn't.

This left the following remaining challenges to be overcome to integrate SQLite into Symbian OS:

- SQLite exposes a C-based API to clients but most Symbian OS programmers would expect a Symbian OS C++ API.
- The databases accessed through SQLite need to comply with features of the operating system such as platform security as well as backup and restore.
- SQLite expects a POSIX API to support it from underneath.

²⁵www.sqlite.org.

Run-time Adaptation

Consider the example of the remote management of a device, for example, a network operator configuring an email account on a subscriber's device 'over the air'. In this case, the provisioning agent or service needs to provide a service to the network that is logically identical to a service provided locally on the device. However, it's often the case that the standards-driven network interface is different to the interface provided locally. Hence a means is needed to join the two interfaces together.

That's not the end of the problem though because clearly it would be beneficial to deploy the same device management system on as many devices as possible and they will vary in their support for features such as email. This means that the central device management system needs to provide an extension point which discovers and loads plug-ins representing the components being remotely managed. This extension point defines a target interface into which we need to adapt existing components.

Solution

This pattern decouples interfaces by creating an intermediary class, called the *adapter*, whose purpose is to adapt the interface presented by one class, the *adaptee*, and present the interface, known as the *target*, expected by another class, the *client*, that wants to use the adaptee. This pattern can therefore be used to solve the problem of incompatibility between two existing interfaces, without requiring changes to either interface.

Structure

From the client's point of view, only the adapter need be visible or known; the implementation of the adapter and any details of the conversion or translation it performs using the adaptee can remain hidden from the client. In short, the client never doesn't directly depend on the adaptee. This is achieved by the structure shown in Figure 9.8.

Dynamics

At its simplest, an adapter is a wrapper. For example, methods are provided that convert the API of a ported application and make it appear to be a native API to clients. Wrappers can also be used to convert outbound system calls from the ported component to match native Symbian OS calls. The adapter simply redirects a call made through the target's interface to a matching adaptee function (see Figure 9.9).

In more complex cases, rather than simply wrapping each method, an adapter may be a complete component that intercepts and converts calls made to one API into calls to a quite different API where there doesn't

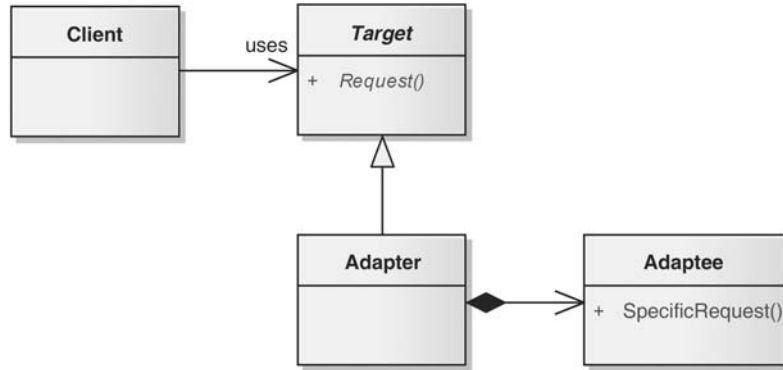


Figure 9.8 Structure of the *Adapter* pattern

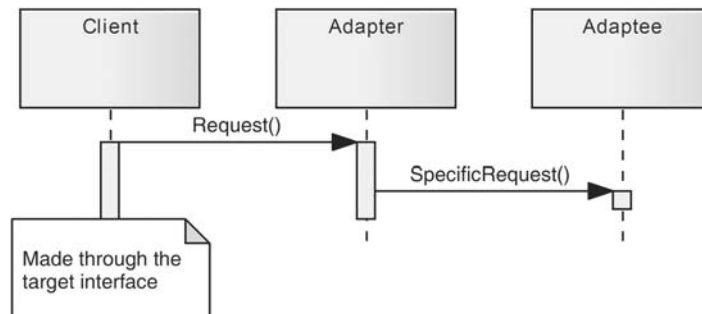


Figure 9.9 Dynamics of the *Adapter* pattern (simple)

exist a close match for the target method called. The adapter may even need to use multiple adaptees to satisfy the client's request on the target interface (see Figure 9.10).

In the most complex case, at run time the adapter may need to select and load the adaptee that satisfies the client's request on the target interface. In these more complex cases, it is common to implement the

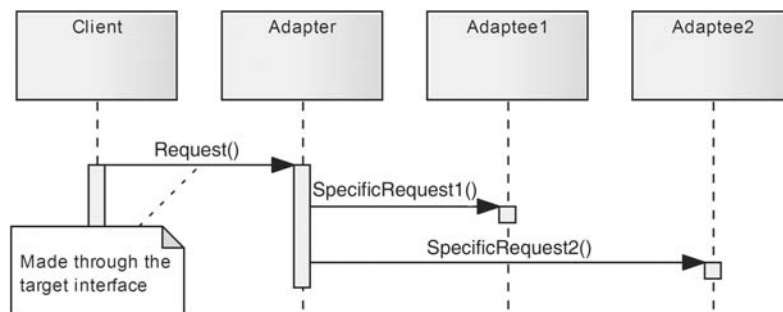


Figure 9.10 Structure of the *Adapter* pattern (complex)

adapter in its own DLL separately from the adaptees so that clients have even fewer dependencies on the adaptees. This allows the adapter to use the ECom service to dynamically load adapters as plug-ins at run time if needed.

Implementation

There's little to say about the implementation of this pattern over and above what's already been said in the Structure and Dynamics sections. The one exception to this is if you wish to load adaptees at run time. If so, you should consider using *Buckle* (see page 252) to do this securely based upon the ECom plug-in service provided by Symbian OS.

Consequences

Positives

- This pattern is simple to implement and understand.
- Development time is reduced by enabling the re-use of existing components.
- By preserving any pre-existing interfaces, you reduce your testing costs because any existing tests written to use the target interface also work for the adaptees if the tests use the adapter.
- It preserves decoupling between the target and adaptee interfaces. By keeping interfaces decoupled, your solution remains flexible which makes future maintenance and evolution easier. Typically, it is much easier to update the adapter in the case of future changes than to try to evolve the clients or the adaptee if that would even be possible.
- In the case of a run-time adapter design, new adaptees can be integrated with potentially no change to the client or even the adapter. This makes your design more open and easily extensible.

Negatives

- It imposes additional overheads on the use of an adaptee interface because an additional layer is interposed between the client and the functionality it wants to use in the adaptee. This increases both the code size and the execution time for the solution although they are unlikely to be significant in all but the most extreme of cases.
- If you are using this pattern to preserve compatibility by maintaining a legacy API in addition to a new API then this necessarily increases the complexity of the system since there are now two APIs for clients to choose from during development as well as two APIs for the provider of the adapter to support.

- Using a run-time adapter based on the ECom plug-in service has no impact on execution time once the plug-in has been loaded. However, searching for and loading a plug-in is an additional overhead compared to statically loading a DLL which is why, if your intention is just to wrap a single legacy component, a regular DLL could be a better choice.

Example Resolved

Preserving Compatibility

In the example described above there was a need to maintain a target interface, `CCommsDatabase`, originally implemented by the `CommDB` component. However this interface needed to work with the new location for the communications database within the Symbian OS Central Repository instead of the old DBMS location. Figure 9.11 shows the old structure of `CommDB`.

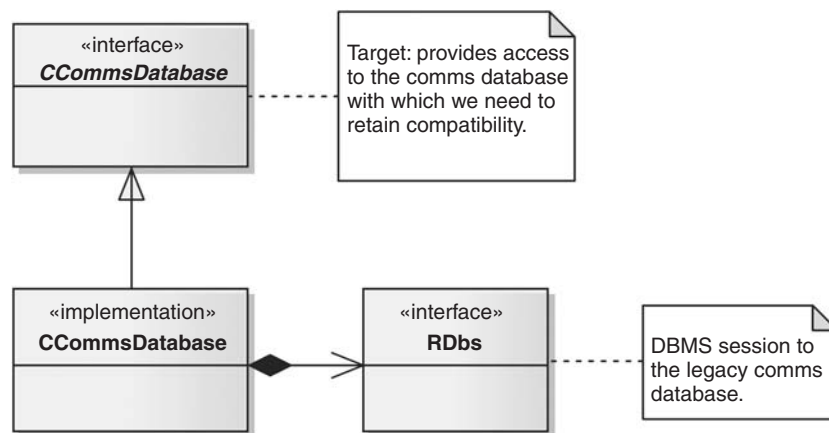


Figure 9.11 Legacy structure of `CommDB`

This problem was solved by removing the old `CommDB` component and providing a replacement, the adapter, called the `CommDB shim` which implements the `CCommsDatabase` interface, the target, in terms of the new communications database component `CommsDat`, the adaptee. This is possible because `CommsDat` provides all the functionality of `CommDB` and more. Figure 9.12 shows the new structure.

This solution clearly provides source compatibility between the legacy and the new shim implementations of the `CCommsDatabase` interface since the same class is used in both. Binary compatibility is also preserved by providing the new adapter in a binary with the same name as used in the legacy solution, `commdb.dll`. However, this doesn't guarantee

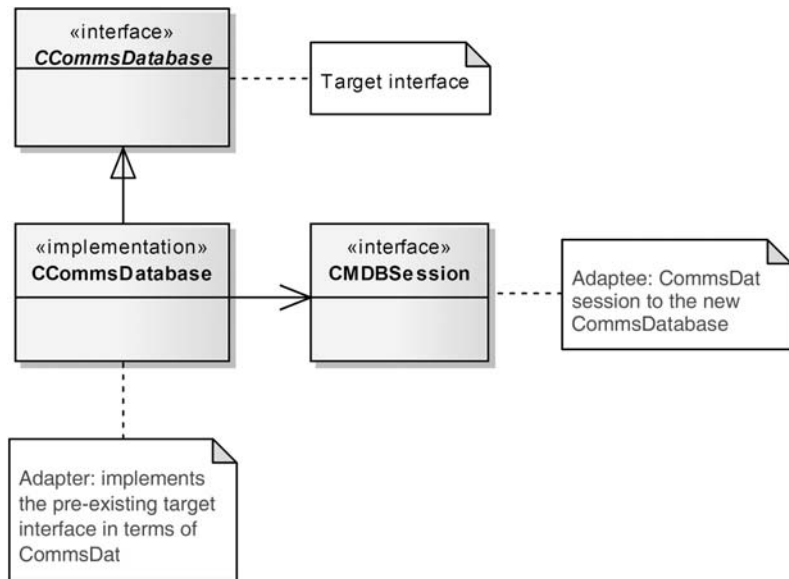


Figure 9.12 Structure of the CommDB Adapter

that the new implementation of the old interface will behave exactly as it used to. The only way to guarantee that this behavioral compatibility has been maintained is to ensure the tests written for the legacy CommDB component work for the new shim version.

The result of this is that Symbian OS now has two APIs to one communications database: CCommsDatabase and CMDBSession.

Wrapping Components to Ease Porting

The example described above for this use of the pattern was the porting of SQLite to Symbian OS. The first problem was that, as SQLite is implemented in standard C, it expects a POSIX API to support it from underneath to provide interfaces into the memory model, the file system, and so on. This was solved by using the PIPS²⁶ to provide near-perfect support for the SQLite C calls, requiring little or no tuning of the SQLite code and working ‘straight out of the box’.

More challenging was providing an interface for use by clients of SQLite. The problem was not that the interface needed to be compatible with the legacy database service, DBMS. Instead the existing interface is written in C and most Symbian OS programmers would expect a Symbian OS C++ API that supports the system-wide functionality, such as platform security as well as backup and restore. So a new interface for Symbian OS, based on C++ was needed.

²⁶PIPS is POSIX on Symbian (PIPS) is itself a large wrapper for a number of native Symbian OS C++ APIs.

The key to understanding the final design is to remember that the new interface to SQLite would have to check the security credentials of any clients attempting to access a database. Since this can only be done securely across a process boundary,²⁷ this immediately rules out the use of *Client-Thread Service* (see page 171) to provide the service interface. Instead, *Client-Server* (see page 182) was selected to be able to provide a secure service interface to the underlying SQLite component. Accordingly the new component is called the SQL Server. This server provides all the support specific to Symbian OS needed by clients accessing the SQLite library underneath. It resulted in the component structure shown in Figure 9.13.

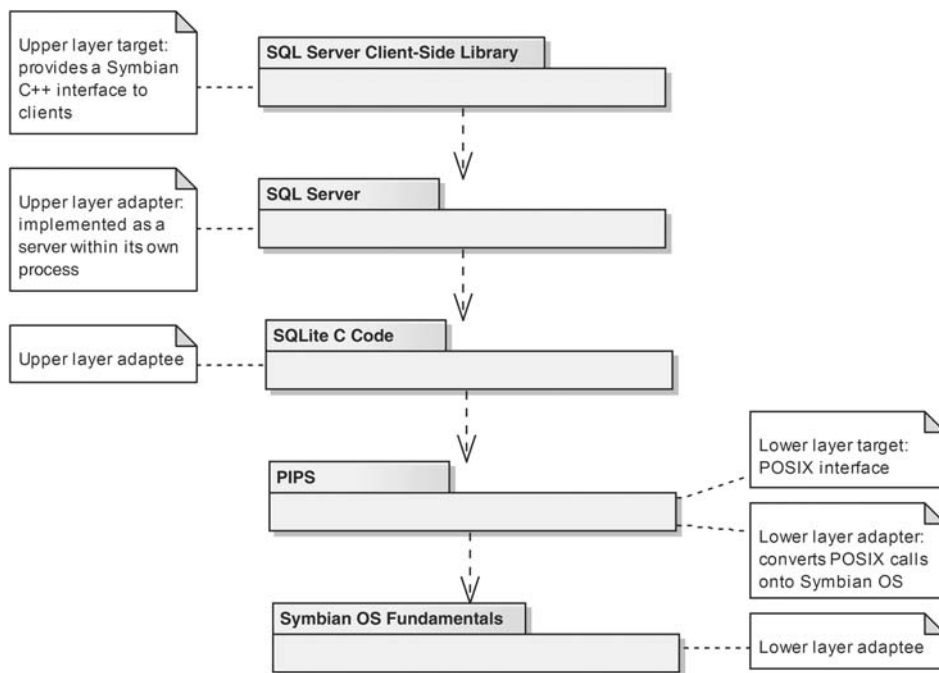


Figure 9.13 Structure of the SQLite Adapter

Run-time Adaptation

Another example of this pattern is provided by the Symbian OS Device Management (DM) subsystem, which supports all aspects of remote provisioning of devices (for example by network operators) to manage device settings. In this case, multiple adapters are required which implement a common provisioning interface `CSm1DmAdapter`, the target, which is used by the DM client on the handset acting as an agent for a DM server

²⁷For an explanation, see the introduction to Chapter 7 in [Heath, 2006].

hosted remotely off the device. The adapters map the common target DM interface into various settings interfaces provided by diverse Symbian OS components, for example phone, messaging, and mail.

This gives rise to the structure shown in Figure 9.14.

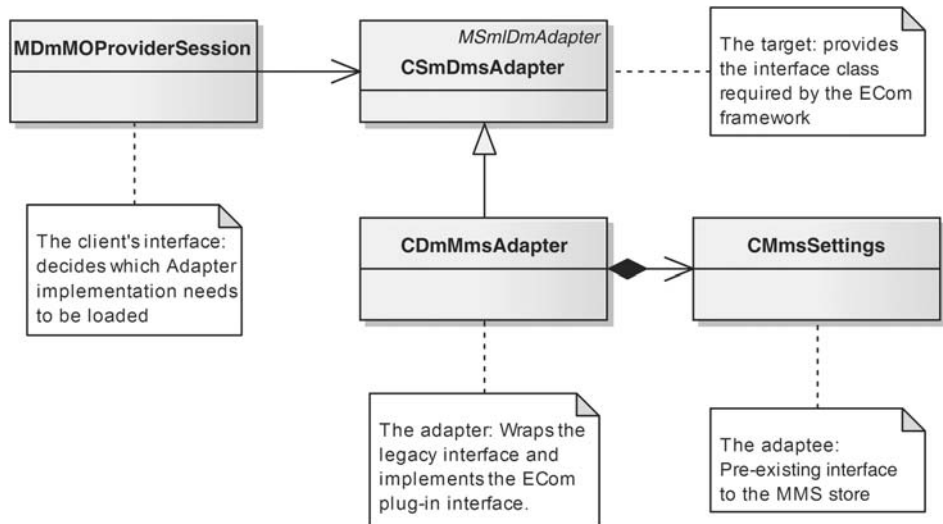


Figure 9.14 Structure of the Device Management Adapter

However, this only partly solves the design problem. Some issues still remain:

1. Shielding the client from changes in the adaptee with a binary firewall (not just a compilation firewall)²⁸ to reduce any dependencies during development.
2. Keeping track of the number of adapters available on the device.
3. Loading the appropriate adapter based on the type of settings it supports.
4. Supporting the addition or removal of adapter implementations to or from the device at run time to allow third-party developers to supply adapters for the DM client.

Here ECom comes to the rescue. By implementing the adapters as ECom plug-ins, the adapters have to be provided in a separate DLL from the DM client, thereby making it completely agnostic of the implementation details or locations of adapters.

²⁸This can be achieved to a certain extent with a regular DLL. However, with DLLs one has to supply the module definition (DEF) file to be used by the 'client' at link time. In this case, clearly such a link-time dependency is not suitable.

Issues 2 and 3 are resolved by ECom's `REComSession::ListImplementationsL()` and `REComSession::CreateImplementationL()` methods, which allow the DM client to specify a 'tag'²⁹ that identifies the plug-ins it is looking for.

Furthermore, the registry of ECom plug-ins is updated whenever a plug-in is added or removed. Clients of the ECom service can register for notification of when this happens, through `REComSession::NotifyOnChange()`. This is used by the DM Client so that it can maintain an up-to-date list of adapter implementations available on the device, which addresses issue 4.

This example also illustrates how adapters can potentially handle more than one adaptee. Going back to the definition of DM, provisioning a device includes operations such as adding, deleting or modifying various device-level settings. Though there are different types of such settings, each one must support these three main operations. The OMA³⁰ DM provisioning standard defines these as ADD, DELETE and REPLACE 'commands' respectively.

Consider the case of remotely managing the MMS accounts on a device. The MMS framework in Symbian uses two classes to store the details of the MMS accounts:

- `CMmsAccounts` is the container for MMS accounts. Each MMS Account item holds the ID and name of the account along with a reference to its MMS Settings object.
- `CMmsSettings` holds all the configuration information that is associated with an MMS account. It includes settings such as proxy and IAP settings, delivery notification flags, image specifications, and so on.

To 'ADD' an MMS account, the remote server sends the details via a SyncML message. The DM framework passes the received details and uses ECom to decide that it should be passed on to the MMS adapter (`CDmMmsAdapter`). `CDmMmsAdapter` in turn has to deal with both `CMmsAccounts` and `CMmsSettings`.

Figure 9.15 shows a simplified version of how the DM MMS adapter distributes work to its multiple adaptees.

When instructed to create a new MMS account, the `CDmMmsAdapter` does the following:

```
// Create a new MMS Settings object
mmsSettings = CMmsSettings::NewL(); // Adaptee 1
CleanupStack::PushL(mmsSettings);
```

²⁹This 'tag' can be anything, ranging from MIME types to number codes. It is specified in the resource (RSS) file of the ECom Plug-in. Custom ECom resolvers can be designed easily to handle the 'tag'. Alternatively, the implementation UID of the plug-in can be used directly.

³⁰www.openmobilealliance.org.

```
// Add this to Settings array
iMmsSettings.AppendL(mmsSettings);

// Ownership transferred
CleanupStack::Pop(mmsSettings);

// Device-management house-keeping code

// Create a new MMS account
iMmsAccounts.CreateMMSAccountL(aName, *mmsSettings); // Adaptee 2
```

This is followed by calls to set (or reset) various MMS Settings parameters (as and when the DM commands arrive), e.g.

```
// Set an MMS Proxy
iMmsSettings[index].AddProxyL(aProxyUid);
```

or

```
// Set delivery notification
iMmsSettings[index].SetAllowDeliveryNotification(aAllow);
```

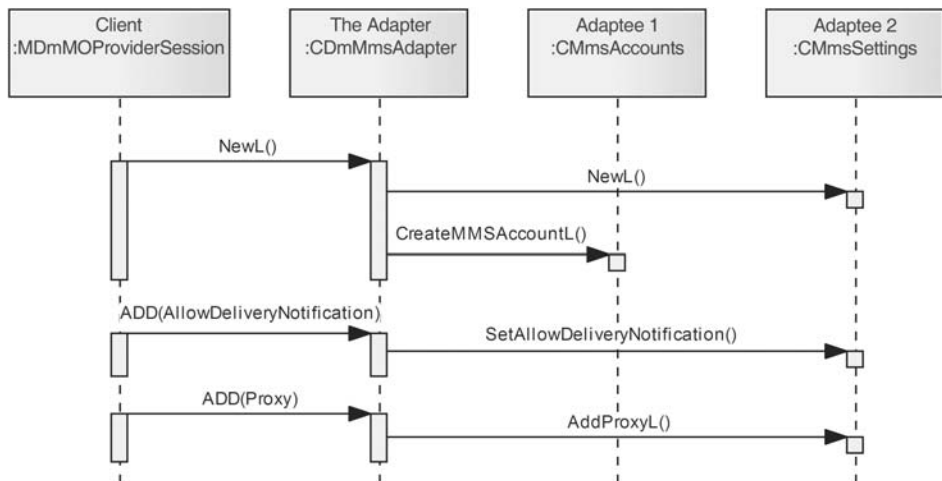


Figure 9.15 Dynamics of the Device Management *Adapter* pattern

Other Known Uses

- *Preserving Compatibility*
A number of features of Symbian OS may be excluded from a device by a device manufacturer, usually because it doesn't have the hardware to support them. For example, this is true of infrared and

Bluetooth. However, if the feature is excluded completely then any existing software that uses the feature would fail to compile as the APIs it depends on aren't present. Instead a mechanism is provided to allow existing software to compile and instead check at run time if the feature is present.

This pattern is used to allow such clients to compile by adapting the target interface of a feature to a non-existent adaptee. The adapter simply provides stubs for the target interface that return `KErr-NotSupported`. This provides a binary- and source-compatible component for clients to use though, of course, it's not possible to provide behavior compatibility!

- *Porting*
Java, Freetype, and OpenGL have all been ported to Symbian OS using this pattern.
- *Run-time Adapters*
SyncML uses ECom-based adapters to implement an extensible framework for handling multiple transport mechanisms. A number of Transport Connection Adapters (TCAs) exist for the various transport types (HTTP, OBEX over USB, OBEX over Bluetooth, WSP, etc.) and are implemented as ECom plug-ins. This allows device manufacturers to create new adapter plug-ins to add support for new transport mechanisms.

Variants and Extensions

None known.

References

- *Buckle* (see page 252) should be used when providing run-time adapters.
- *Handle-Body* (see page 385) also aims to isolate a component's implementation from the client. However, the chief intent of a *Handle-Body* is to allow the client interface and a component implementation to vary independently. Also, this pattern is typically used to make existing, unrelated classes work together; both parts of *Handle-Body* are typically introduced together during development to connect classes which are part of the same design.
- Proxy [Gamma *et al.*, 1994], like Adapter, 'stands in' for a component. A Proxy, however, provides the same interface as the actual component unlike an Adapter.
- Façade [Gamma *et al.*, 1994] 'wraps' the complexity of various, often dissimilar (though logically cohesive), interfaces by combining them into a unified, easy-to-use interface.

Handle–Body

Intent Decouple an interface from its implementation so that the two can vary independently.

AKA Cheshire Cat

Problem

Context

One of your components needs to expose an interface for use by external clients.

Summary

- You wish your component’s interface to be as encapsulated as possible.
- You wish to maintain the compatibility of the exposed interface across multiple versions of your component whilst retaining the freedom to change how it is implemented.
- You would like to rapidly develop a component but also allow for optimization or the addition of new features later.

Description

As the software in mobile devices becomes ever more complicated, it also becomes more difficult to both maintain and extend it.

You will often find yourself asking questions such as:

- Does my software implement a specification that is subject to changes and updates?
- Will I need to allow a different implementation to be provided?
- Do I need to cope with different versions of software?
- Do I need to develop an early version of a component as soon as possible which is ‘good enough’ now but will need to be re-factored later on?

The problems of developing software that is easy to expand, maintain and re-factor has and always will be a key issue that you will face. A standard way to achieve this is to ensure your interfaces are well encapsulated and don’t ‘leak’ any unnecessary details or dependencies

especially when external clients are relying on the interface to remain stable and not change in any way that will stop them from working correctly.

But how can we do this in C++? There are a number of ways to accidentally break clients. For instance, you'd think that adding a private data member to a class would be safe since it's private after all. However, doing this increases the size of your class and any client code that calls a class constructor relies on that being fixed. Indeed one of the benefits of the second-phase construction common in Symbian OS³¹ is that clients don't get to call the constructor directly. Instead they get passed a pointer to the class and hence don't depend on the size of the class which allows it to be changed. Unless it's an abstract class and clients are expected to derive from it – then they have to call your class's constructor which makes them sensitive to its size again.

Whether you are developing low-level kernel modifications, middle level services or end-user applications, the ability to be able to change software quickly and easily will make your life as a developer much less stressful!

Example

The HTTP framework provided by Symbian OS supports HTTP requests and HTTP responses. A simple design would implement these as two distinct classes as conceptually they are used for different purposes. One factor influencing the design would be that the requests and responses form part of a well-known protocol that is unlikely to undergo any major changes.

The framework needed to be developed in an iterative fashion so that clients had at least some HTTP functionality as soon as possible. Additional features were then added later. For example, the very first version of the HTTP framework only supported HTTP GET. It would take a further nine months and five iterations to fully develop a framework which had all the required functionality that a web browser requires.

Although the two-phase construction idiom used in Symbian OS means that the size of the classes could be increased without breaking compatibility, it was felt that this still exposed too much of the underlying implementation details to the clients. If one mistake was made that meant clients accidentally relied on the details of the implementation rather than the interface this would've caused serious problems down the line. A way of further separating the implementation of the classes from its interface was needed.

It was the uncertainty of the future design and the realization that it would almost certainly need to change that lead to the use of this pattern

³¹No doubt you'll have seen this many times as most C classes have a `NewL()` factory function in Symbian OS. See also [Harrison and Shackman, 2007, Section 4.5].

not just for the request and response classes but across the entire HTTP framework.

Solution

This pattern supports and encourages encapsulation and loose coupling by hiding the implementation details from your clients by dividing one class into two. The first of these, the *handle*, simply defines the interface and then defers all implementation of that interface to the *body*. One of the first descriptions of this was in [Coplien, 1991].

A point to remember is that the word 'handle' being referred to here does not necessarily imply the use of a Symbian OS handle derived from `RHandleBase` such as `RThread` or `RProcess`.

Structure

A client is only ever able to see the interface class and just uses that directly. However, the interface class delegates all calls through a pointer to the underlying handle which provides the implementation as shown in Figure 9.16.



Figure 9.16 Structure of the *Handle-Body* pattern

This particular technique is often referred to in C++ as the 'Cheshire Cat Idiom', named after the character from Lewis Carroll's *Alice in Wonderland*, and refers to the point where the cat disappears gradually until nothing is left but its smile. Here the smile is the implementation pointer.

If a data member is added to the body class then the size of the handle class remains the same since its size is just that of a pointer to the body. Hence the client does not need to be recompiled as it is isolated from this change.

The handle and the body classes are normally defined in the same DLL but only the handle is exported. This *binary firewall* allows the client to be shielded from any changes in the body. This is particularly advantageous if you would like to vary the details of the implementation depending on the situation. For instance, if the interface is used to store and retrieve objects, the implementation might need to change depending on exactly how many objects you expect clients to store. This allows your interface to provide the best performance for each situation.

Dynamics

The dynamics of this pattern aren't very complicated as the handle simply forwards all calls to its body as shown in Figure 9.17.

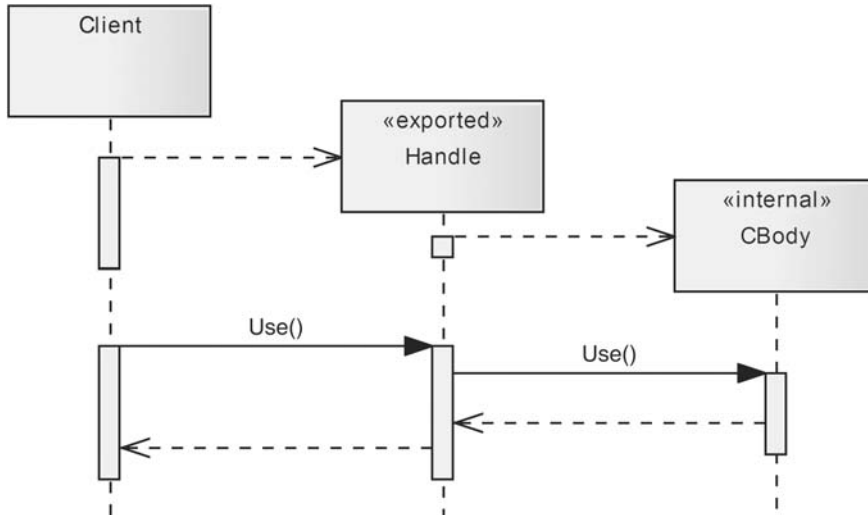


Figure 9.17 Dynamics of the *Handle-Body* pattern

Note that we assume that the body is created at the same time as the handle which, if we consider the body to be a resource, means *Immortal* (see page 53) is being used. This isn't necessarily the case and other lifetimes can be chosen for the body as described in Chapter 3.

There is one additional subtlety that should be mentioned though and that's the fact that by forcing the body to be allocated on the heap you *may* have introduced a new error path for the client to deal with. Consider for a moment that you weren't using this pattern and instead provided an exposed interface as a T class used solely on the client's stack. Changing the design to use a pointer to a body class instead would mean that clients now also have to deal with the handle possibly failing to be created which wasn't a problem previously.

Implementation

To use this pattern you need to separate your interface from your implementation. If you have an existing class that you wish to convert to using this pattern then you need to scoop out all the private data members, private functions and implementation of the original class and put them to one side for now. If you add a pointer to a CBody class,

which we'll define later, then you have the beginnings of your handle class.

You then have the choice of implementing your handle as either an R or a C class.³² For this illustration, we've chosen to use a C class to avoid obscuring the main thrust of the pattern with the additional considerations that you need to take into account when using an R class.

Next you should forward declare your body class in the header file used to define your handle class. This avoids you having to `#include` any header files for the body and is another means of ensuring the interface is independent of its implementation. It also incidentally reduces the amount of code a client needs to compile when using the interface.

This gives the following header file for the handle class:

```
class CBody;

class CHandle : public CBase
{
public:
    IMPORT_C static CHandle* NewL();
    IMPORT_C void DoSomethingL();
private:
    CHandle();
private:
    CBody* iBody
};
```

The source file for the interface simply has to forward any calls on to the body.

```
EXPORT_C void CHandle::DoSomethingL()
{
    iBody->DoSomethingL();
}
```

Some additional points to note include:

- No handle functions should be declared as inline. Whilst it is tempting to do this as their implementation is so simple, it means you will not be able to change the function ever again without breaking binary compatibility. This is because an inline function puts the code for the function in the client's binary³³ and not yours, which breaches the binary firewall that we are trying to erect.

³²It can't be an M class as it has a data member, nor can it be a T class as the data is on the heap.

³³Whilst it's true that sometimes functions marked as inline aren't actually inlined by the compiler, it can happen and so we have to treat the functions as if they will actually be inlined to avoid any risk of breaking compatibility.

- The handle class normally does not contain any private functions except for the constructor functions. Instead all such functions should be implemented by the body class.
- Do not implement the body by providing virtual functions in the handle class and then deriving the body from the handle. Whilst this makes the implementation of the handle slightly simpler, it effectively destroys the benefits of this pattern by exposing the size of the class to further derived classes; introducing vtables which are tricky to maintain binary compatibility for, and preventing alternative bodies to be used at run time.
- No `ConstructL()` function is needed as normally you would just call a function on the body that returns a fully constructed object.

Consequences

Positives

- This pattern allows an interface to be extended without breaking pre-existing clients.
Over time, you will need to extend the interface to add new functionality. This pattern gives you more flexibility when you need to extend your component because it removes restrictions that you would otherwise have felt. For instance, you are completely free to change the size of the implementation class even if the interface has been derived from. Most changes to the implementation are hidden from clients.
- This pattern reduces the maintenance costs for your component by allowing an implementation to be replaced or re-factored without impacting clients of your interface.
Over time, code becomes more difficult to maintain as new features are added. At some point it becomes more beneficial to completely replace, re-factor or redesign the existing code to allow it to perform better or to allow for new functionality to be added. Where this pattern has been used, this is significantly easier to do because the interface is well encapsulated so that as long as the interface remains compatible, the underlying implementation can be completely re-factored or replaced.
- You have the freedom to choose a different implementation.
Since this pattern hides so much of the implementation from clients of your interface you have more freedom to select the most appropriate body for the circumstances. This can be done at compile time but also at run time for even more flexibility. If you do need to dynamically load a body then you should use the Symbian OS ECom plug-in service.³⁴ Remember that, in either case, each of the different bodies

³⁴See [Harrison and Shackman, 2007, Section 19.3] for details about ECom.

needs to provide the same overall behavior as it's no good if clients still compile but they then break at run time because your interface performs in ways the client didn't expect.

- Your testing costs are reduced.
Since the interface to your component is well encapsulated you will have fewer side-effects to deal with during testing of the component using this pattern. However, this pattern also makes it easier to create mock objects³⁵ when testing other components by making it simple to replace the real implementation with a test implementation.

Negatives

- Additional complexity can increase maintenance costs.
If you hadn't applied this pattern then you'd have one fewer class in your design. Whilst this doesn't sound like much, if you apply this pattern to a lot of interfaces then you can get into a situation where class hierarchies can be quite complicated. This can make it harder to understand and debug your component as a result.
- Additional overheads are caused by the extra indirection.
By introducing an additional level of indirection this incurs small execution-time and code-size penalties for your component. The extra function call used when calling the body class, and the extra object allocation, leads to the loss of some CPU cycles whilst the extra class requires more code.

Example Resolved

As seen in the solution above this is a very simple pattern, but it is always worth looking at how it has been used in a real example.

As described above, Symbian OS provides an interface for clients to use HTTP functionality. One of the interfaces exposed by this component is the `RHTTPSession` class which represents a client's connection with a remote entity and allows multiple transactions to take place using this connection. This class is the starting point for developers using the Symbian OS HTTP service.

Since it was important to provide a stable interface to clients whilst not restricting future growth, this pattern was used.

Handle

`RHTTPSession` is the handle as it provides the interface used by clients. In this case it is an R class to indicate that it provides access to a resource owned elsewhere, which in this case is in the HTTP protocol itself.

³⁵en.wikipedia.org/wiki/Mock_object.

The following code shows the most important points of this class from the point of view of this pattern. The full details can be found in `epoc32\include\http\rhttpsession.h`.

```
class CHTTPSession; // Note the forward declare of the body

class RHTTPSession
{
public:
    IMPORT_C void OpenL();
    IMPORT_C RHTTPHeaders RequestSessionHeadersL();
    ...
private:
    friend class CHTTPSession;

    CHTTPSession* iImplementation; // Note the pointer to the body
};
```

The following is the simplest of several different options for beginning an HTTP session. This method, `OpenL()`, acts as the factory construction function for `RHTTPSession` and allows clients to create and initialize a session with the HTTP service.

This function call constructs the body by simply calling its `NewL()`. Slightly unusually, the handle function, `OpenL()`, has a different name to this body function. This simply reflects the fact that the handle is an R class whilst the body is a C class which means developers wouldn't expect to see a `NewL()` on `RHTTPSession` whilst they would on `CHTTPSession`. The functions also act very slightly differently in that `OpenL()` is called on an existing object whilst `NewL()` is a static function and doesn't need to be called on an object.

Finally, note the use of *Fail Fast* (see page 17) to catch clients who call this function twice which would otherwise cause a memory leak.

```
EXPORT_C void RHTTPSession::OpenL()
{
    __ASSERT_DEBUG(iImplementation == 0,
        HTTPPanic::Panic(HTTPPanic::ESessionAlreadyOpen));
    iImplementation = CHTTPSession::NewL();
}
```

The following shows an example of one of the handle's functions that simply forward the call on to the body to deal with.

```
EXPORT_C RHTTPHeaders RHTTPSession::RequestSessionHeadersL()
{
    return iImplementation->RequestSessionHeadersL();
}
```


Body

The `CHTTPSession` class forms the body in this use of the pattern as follows:

```
class CHTTPSession : public CBase
{
public:
    static CHTTPSession* NewL();
    RHTTPHeaders RequestSessionHeadersL();
    inline RHTTPSession Handle();
    ...
private:
    CHeaderFieldPart* iConnectionInfo; // The proxy or connection details
    RArray<CTransaction*> iTransactions; // All the open transactions
    RArray<THTTPFilterRegistration> iFilterQueue;
    ...
};
```

The `NewL()` isn't shown because it is implemented in the standard Symbian way as a two-phase constructor. However, here is the implementation of `CHTTPSession::RequestSessionHeadersL()` which provides the functionality for `RHTTPSession::RequestSessionHeadersL()`.

Note the use of *Lazy Allocation* (see page 63) to create the request header collection class on demand. Apart from that and returning an object on the stack from the function, there is nothing unusual here.

```
RHTTPHeaders CHTTPSession::RequestSessionHeadersL()
{
    // Create the session headers if necessary
    if (iRequestSessionHeaders == NULL)
    {
        iRequestSessionHeaders = CHeaders::NewL(*iProtocolHandler->Codec());
    }
    return iRequestSessionHeaders->Handle();
}
```

The one additional function we should discuss is the `Handle()` function provided by the body which returns an `RHTTPSession` instance which uses the current object as its body. Providing a `Handle()` function on a body class may look slightly odd at first but points to an interesting side-effect which the HTTP framework takes full advantage of.

If you take a closer look at `RequestSessionHeadersL()`, you will see that it returns an object by value rather than a reference or pointer to one. This is because `RHTTPHeaders` is also implemented using this pattern and is a handle to a `CHeaders` object. This means

that `RHTTPHeaders` is an object of size 4 bytes, because of its `iBody` pointer, which makes it efficient to pass by value on the stack.

The `CHTTPSession::Handle()` method works in the same way and creates a new handle to itself. The function can be inlined because it's only ever used within your DLL and so there's no danger of it leaking out into client DLLs and causing compatibility problems in the future. If you were wondering why `CHTTPSession` was declared as a friend by `RHTTPSession` it's so the following would be possible:

```
inline RHTTPSession CHTTPSession::Handle()
{
    RHTTPSession r;
    r.iImplementation = this;
    return r;
}
```

This isn't an essential part of implementing this pattern but can be useful in other implementations as you can use the handle class in a similar way to the way you would treat a pointer to the body class but with more protection. Note that this all works because only shallow copies of the handle class are made. Hence copying a handle could mean that there are two handles pointing to the same body just as if you'd copied a pointer. This means the ownership of the handle classes should be treated carefully to avoid double deleting the body.

Other Known Uses

- *Image Conversion*
This component makes use of this pattern in both `CImageDisplay` and `CImageTransform` to provide these widely used classes with the flexibility to be extended in future to allow different image formats to be displayed and transformed through the same interface by simply changing the underlying body providing the implementation of the interface.
- *ECom*
The Symbian OS ECom plug-in service is based on this pattern as frameworks interact with plug-ins through a handle class. A plug-in provider then provides a body class to implement that handle.

Variants and Extensions

- *Rapid Development using Handle–Body*
Let's consider interface design. How often have you sat down and tried to come up with a design for a class's interface without thinking about its implementation? Never? Well, you are in good company.

It turns out that most developers actually consider multiple different implementations when designing an interface. This is a good thing. It is even better if you can prototype several implementations. You can use the *Handle–Body* pattern to ensure that the same interface is provided by all your prototype implementations and also make sure that the interface is not dependent on an implementation.

If you write all of your test code to work through your interface this also allows you to use it to test each of the different prototype implementations. Using this pattern can also allow you to delay your implementation, for example you could provide the interface and a prototype or stub implementation to a customer knowing that the code they write will work with your implementation when you finish it. In the world of mobile phone operating system development this can be a key factor in reducing the time to market of a phone with your new software in it.

References

- [Coplien, 1991] discusses *Handle–Body* in a different context.
- *Adapter* (see page 372) is related to this pattern in that the client uses an interface that doesn't directly provide the implementation but instead forwards calls to another object. A key distinction between the two is that here the handle and body classes are provided by the same vendor whilst this is not the case with *Adapter*.
- See also Chapter 3 for some ways to deal with the lifetime of the body object.
- This pattern is a specialization of Bridge [Gamma *et al.*, 1994] in which there can be whole hierarchies of handle classes joined to another hierarchy of body classes.
- *Façade* [Gamma *et al.*, 1994] is a similar pattern to this one but differs in that it allows the interface to change between the handle and the body.

Appendix A

Impact Analysis of Recurring Consequences

This appendix analyzes the impact of consequences that recur in the patterns described in this book. Note that the data given here is only intended to allow you to assess the feasibility of design decisions and should not be taken as accurate figures to be precisely relied upon. We recommend that you measure directly any facts on which you base the development of your software.

In this discussion, the following factors can make a significant difference to the impact of using a pattern:

- which type of CPU is present on the device
- which memory model is being used: this is determined by the CPU;¹ ARMv5 usually implies the moving memory model whilst ARMv6 usually implies the multiple memory model
- whether code is eXecuted In Place (XIP) or not (non-XIP).

At the time of writing, the most common device configuration is an ARMv6 core with non-XIP flash. However, older or midrange devices may well use an ARMv5 core with either XIP or non-XIP flash.

See also [Sales, 2005] as this provides helpful background material to the discussion here.

Run-time Impact of a New Thread

RAM

The RAM impact of a new thread can change depending on the size of the user stack it is given, as well as the size of the heap it is using. Hence

¹This may be listed on the device manufacturer's website, e.g. for S60 devices: www.forum.nokia.com/devices.

it is only possible to give an absolute minimum figure for RAM usage. A thread uses RAM for the following things:

- supervisor mode call stack – fixed at 4 KB
- user mode call stack – absolute minimum of 4 KB if steps are taken to reduce it from the Symbian OS default of 8 KB; some UI vendors raise the default stack size to 20 KB or more and may prevent it from being reduced below this value
- heap – absolute minimum of 4 KB but can be shared between threads if required
- objects used by the Kernel to track the thread – approximately 1 KB.

Hence the absolute minimum RAM used by a new thread is 9 KB if the user mode stack is reduced to its minimum and the thread shares the heap of an existing thread. However, the default minimum is at least 17 KB.

Thread Creation

The time taken to create a thread will vary according to how busy the system is, which CPU is present on the device, etc. However to give you some idea of the time needed to create a new thread within an existing process here are some rough figures measured at the time this was written:

- OMAP1623/ARM9/ARMv5 UREL – 600 microseconds
- OMAP2420/ARM11/ARMv6 UREL – 300 microseconds

Inter-thread Context Switching

The time to switch between two threads in the same process varies according to how busy the system is, which CPU is present on the device, etc. However to give you some idea of the time taken here are some rough figures measured at the time this was written:

- OMAP1623/ARM9/ARMv5 UREL – 2 microseconds
- OMAP2420/ARM11/ARMv6 UREL – 2 microseconds

Note that this is effectively the time needed to pass a message between the two threads.

Run-time Impact of a New Process

RAM

The RAM impact of a new process depends on the type of smartphone you are targeting since this determines the following key factors:

- whether the code executed by the process is run from XIP or non-XIP flash
- which memory model is being used.

The RAM used by a process also includes the RAM used by at least one thread since a process cannot exist without a main thread. As discussed above, a thread uses an absolute minimum of 9 KB. However, this assumed there was another thread to share a heap with. If there is only a single thread in a process this isn't possible and the absolute minimum RAM used *by a single thread* is 13 KB. The default RAM usage for a thread isn't affected by this and so is still at least 17 KB.

A process also uses RAM for a number of other factors such as the amount of code loaded into the process, additional kernel objects, page tables, etc. This all adds up to the following absolute minimum RAM usage shown in Table A.1.

Table A.1 RAM Used by a Process

	XIP Code		Non-XIP code	
	Absolute Minimum	Default Minimum	Absolute Minimum	Default Minimum
Moving memory model	17 KB	At least 21 KB	21 KB	At least 25 KB
Multiple memory model	25 KB	At least 29 KB	30 KB	At least 34 KB

Memory Model Limitations

An additional concern when creating a new process is that there is a fixed limit on how many processes can be running at the same time. On the moving memory model, the limit is 1000 simultaneously running processes, while on the multiple memory model it is 256.

Process Creation

The time taken to create a process varies according to how busy the system is, which CPU is present on the device, etc. However to give you some idea of the time needed to create a new process, here are some rough figures measured at the time this was written:

- OMAP1623/ARM9/ARMv5 UREL – 20,000 microseconds
- OMAP2420/ARM11/ARMv6 UREL – 10,000 microseconds

Inter-process Context Switching

The time to switch between two processes varies according to how busy the system is, which CPU is present on the device, etc. However to give

you some idea of the time taken here are some rough figures measured at the time this was written:

- OMAP1623/ARM9/ARMv5 UREL – 70 microseconds
- OMAP2420/ARM11/ARMv6 UREL – 10 microseconds

Note that this also gives a minimum time for a message passed between the two processes. However, if a large amount of data is transferred (and therefore copied), the time needed for that could be more significant than the time to simply switch contexts.

Run-time Impact of Increased Code Size

Increasing the code size of an executable can impact the following system characteristics:²

- The executable takes up additional disk space.
- More RAM is needed to load the executable.
- Loading the executable takes longer.

However, which characteristics are impacted and to what degree varies according to how the device is configured and what technologies it uses. Table A.2 summarizes the determining factors.

Table A.2 Factors Affecting the Run-time Impact of Increased Code Size

Potential Impacts of Increased Code Size	Device Configuration			
	XIP	Non-XIP		
		Moving Memory Model	Multiple Memory Model	Code is Demand Paged
Disk space	Particularly impaired	Impaired	Impaired	Impaired
RAM usage	No significant impact	Increases	Increases independently of how many processes use the code	Reduces impact of the memory model
Time to load the code	No significant impact	Impaired	Impaired	Variable (see below)

²Characteristics, such as maintainability, testability, etc., can also be influenced by having more code but they’re not significantly influenced by how the device is configured.

As you can see the most important factor is whether the code is loaded from XIP or non-XIP flash. If it's on XIP flash then the Loader has to do very little to enable the code to be executed. If the code is on non-XIP flash then, by definition, the code cannot be executed in place and the Loader has to allocate RAM in which to hold the code that will be executed. In addition, it's not a simply case of copying the code from flash to RAM; the code may need to be decompressed in addition to having all of its implicit linkages resolved. Note that because code can be compressed whilst stored on disk this means XIP code has a relatively bigger impact on disk space than non-XIP code.

For code stored on non-XIP flash, the different memory models impact the RAM used to hold the code to be executed in different ways. The multiple memory model copies each DLL into RAM once and then each process references a single copy of the code. However, this isn't true for the moving memory model, in which multiple copies may be loaded although only if absolutely necessary.³ Hence the multiple memory model will always use the same or less RAM than the moving memory model when representing executable code at run time.

Another factor affecting code stored on non-XIP flash is demand paging⁴ which can make a big difference by reducing the impact of increased code size on RAM usage and the time to load code. This is because, prior to Symbian OS v9.3, entire DLLs are copied into RAM when they are needed; when demand paging is enabled only the required 'page' within the DLL is loaded into RAM when a reference is made to it (this is known as 'paging on demand'). This can significantly reduce the code loaded at any one time and hence the corresponding RAM usage. The effect on the time to load the code is less easy to quantify. Certainly it is common for processes to load faster because less code is loaded all at once. However, when demand paging isn't enabled code is loaded once and then no further time is spent loading code. With demand paging enabled, code can be paged in at any point and hence impact performance at any point. Note that on Symbian OS v9.3, only code built into the device can be demand paged but on Symbian OS v9.5 code that is installed onto the device can also be affected.

³For example, when a DLL uses writeable static data or is loaded into multiple processes in such a way that the data cannot be at the same address in all those processes, such as for a fixed process.

⁴www.symbian.com/symbianos/demandpaging and www.iqmagazineonline.com/article.php?issue=23&article_id=706.

References

- Alexandrescu, A. (2001) *Modern C++ Design: Generic programming and design patterns applied*. Addison-Wesley Professional.
- Babin, S. (2007) *Developing Software for Symbian OS: A beginner's guide to creating Symbian OS v9 smartphone applications in C++*, Second Edition. Symbian Press. See [**developer.symbian.com/books**](http://developer.symbian.com/books) for more information and a sample chapter.
- Ball, S. and Crawford, J. (1997) 'Monostate Classes: The power of one'. *C++ Report*, May 1997. Reprinted in R. C. Martin (ed.), 2000, *More C++ Gems*, Cambridge University Press.
- Berger, E.D., Zorn, B.G. and McKinley, K.S. (2002) 'Reconsidering custom memory allocation', in *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, 1–12. ACM Press New York, NY, USA.
- Blakley, R., Heath, C. and members of The Open Group Security Forum (2004) 'Security Design Patterns', Technical Guide. The Open Group. Available at [**www.opengroup.org/pubs/catalog/g031.htm**](http://www.opengroup.org/pubs/catalog/g031.htm).
- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996) *Pattern-Oriented Software Architecture Volume 1: A system of patterns*. John Wiley & Sons.
- Campbell, I., Self, D., Howell, E., Bunning, I., Rahman, I., Caffery, L., Box, M., Elliott, M., Ho, N., Cochart, P., Howes, T. and Davies, T. (2007) *Symbian OS Communications Programming*, Second Edition. Symbian Press. See [**developer.symbian.com/books**](http://developer.symbian.com/books) for more information and a sample chapter.
- Cooper, J.W. (2000) *Java Design Patterns: A tutorial*. Addison-Wesley.
- Coplien, J.O. (1991) *Advanced C++ Programming Styles and Idioms*. Addison-Wesley.

- Coplien, J.O. and D.C. Schmidt (eds) (1995) *Pattern Languages of Program Design*. Addison-Wesley.
- Densmore, S. (2004) 'Why Singletons are Evil'. Available at blogs.msdn.com/scottdensmore/archive/2004/05/25/140827.aspx.
- DoD (1985) 'Trusted Computer System Evaluation Criteria'. US Department of Defense.
- EU (2002) 'Universal Service and Users' Rights Relating to Electronic Communications, Networks and Services'. (Universal Service) Directive 2002/22/EC. European Parliament. Available at www.legi-internet.ro/universalservice.htm.
- Fagan, M.E. (1976) 'Design and Code Inspections to Reduce Errors in Program Development'. *IBM Systems Journal*, 15(3):182–211. Available at www.research.ibm.com/journal/sj/153/ibmsj1503C.pdf.
- FCC (2001) 'Enhanced 911: Wireless Services'. Federal Communications Commission. Available at www.fcc.gov/pshs/services/911-services/enhanced911/Welcome.html.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J.M. (1994) *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional Computing Series.
- Harrison, N.B., Foote, B. and Rohnert, H. (eds) (2000) *Pattern Languages of Program Design 4*. Addison-Wesley.
- Harrison, R. and Shackman, M. (2007) *Symbian OS C++ for Mobile Phones*, Volume 3. Symbian Press. See developer.symbian.com/books for more information and a sample chapter.
- Heath, C. (2006) *Symbian OS Platform Security: Software development using the Symbian OS Security Architecture*. Symbian Press. See developer.symbian.com/books for more information and a sample chapter.
- ISO 9126 'Software engineering: Product quality'. Available at www.issco.unige.ch/projects/ewg96/node13.html.
- Kienzle, D., Elder, M., Tyree, D. and Edwards-Hewitt, J. (2002) *Security Patterns Repository*. Available at www.scrypt.net/~celer/securitypatterns/repository.pdf.
- Kircher, M. and Jain, P. (2004) *Pattern-Oriented Software Architecture, Volume 3: Patterns for resource management*.
- Knuth, D. (1974) 'Structured Programming with go to Statements'. *ACM Journal Computing Surveys*, 6(4):268. Available at pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf.
- Lavender, R.G. and Schmidt, D.C. (1996) 'Active Object: An object behavioral pattern for concurrent programming', in *Pattern Languages of Program Design*, J. O. Coplien, J. Vlissides, and N. Kerth (eds). Addison-Wesley.
- Longshaw, A. and Woods, E. (2004) 'Patterns for Generation, Handling and Management of Errors'. Available at www.eoinwoods.info/doc/europlop.2004.errorhandling.pdf.

- Martin, R.C., Riehle, D. Buschmann, F. (eds) (1998) *Pattern Languages of Program Design 3*. Addison-Wesley.
- Meyer, B. (1992) 'Applying "Design By Contract"'. *Computer*, 25(10):40–51. IEEE. Available at se.ethz.ch/~meyer/publications/computer/contract.pdf.
- Morris, B. (2007) *The Symbian OS Architecture Sourcebook: Design and evolution of a mobile phone OS*. Symbian Press. See developer.symbian.com/books for more information and a sample chapter.
- Morris, B. (ed.) (2008) 'A Guide to Symbian Signed'. Available at developer.symbian.com/main/support/signed.
- Myers, S. and Alexandrescu, A. (2004) 'C++ and the Perils of Double-Checked Locking'. *Dr Dobbs Journal*. Available at www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf.
- OMA (2002) 'SyncML Data Synchronization Specifications' Version 1.1. Available at www.openmobilealliance.org/tech/affiliates/syncml/syncmlindex.html.
- Rainsberger, J.B. (2001) 'Use your singletons wisely'. Available at www.128.ibm.com/developerworks/webservices/library/co-single.html.
- Rosenberg, R. and Arshad, J. (2007) 'Symbian OS System Model'. Available at developer.symbian.com/main/oslibrary/sys_models/downloads.jsp.
- Sales, J. (2005) *Symbian OS Internals: Real-time Kernel Programming*. Symbian Press. See developer.symbian.com/books for more information and a sample chapter.
- Saltzer, J. and Schroeder, M. (1975) 'The Protection of Information in Computer Systems'. IEEE.
- Saumont, P-Y. (2007) 'Do we really need singletons?'. Available at www.javalobby.org/java/forums/t91076.html.
- Schmidt, D.C. (1999) 'Asynchronous Completion Token: An object behavioral pattern for efficient asynchronous event handling'. Available at www.cs.wustl.edu/~schmidt/PDF/ACT.pdf.
- Schmidt, D. and Harrison, T. (1996) 'Reality Check'. *C++ Report*, March 1996. Available at www.cs.wustl.edu/~schmidt/editorial-3.html.
- Shackman, M. (2006) 'Platform Security: A Technical Overview'. Available at developer.symbian.com/main/downloads/papers/plat_sec_tech_overview/platform_security_a_technical_overview.pdf.
- Silvennoinen, J. (2007) 'Y-Browser'. Available at www.drjukka.com/YBrowser.html.
- Stichbury, J. (2004) *Symbian OS Explained: Effective C++ Programming for Smartphones*. Symbian Press. See developer.symbian.com/books for more information and a sample chapter.

- Stichbury, J. and Jacobs, M. (2006) *The Accredited Symbian Developer Primer*. Symbian Press. See [**developer.symbian.com/books**](http://developer.symbian.com/books) for more information and a sample chapter.
- Vlissides, J. (1996) 'To kill a singleton'. *C++ Report*, June 1996. Available at [**www.research.ibm.com/designpatterns/pubs/ph-jun96.txt**](http://www.research.ibm.com/designpatterns/pubs/ph-jun96.txt).
- Vlissides, J., Coplien, J.O., Kerth, N.L. (eds) (1996) *Pattern Languages of Program Design 2*. Addison-Wesley.
- Weir, C. (1998) 'Code that tests itself: Using conditions and invariants to debug your code'. *C++ Report*, March 1998. Available at [**www.charlesweir.com/papers/selftest.pdf**](http://www.charlesweir.com/papers/selftest.pdf).
- Weir, C. and Noble, J. (2000) *Small Memory Software*. Addison-Wesley Professional.
- Wilden, L.H. et al. (2003) *OggPlay*. Available at [**symbianoggplay.sourceforge.net**](http://symbianoggplay.sourceforge.net).
- Willee, H. (Nov 2007) 'Support for Writeable Static Data in DLLs'. Available at [**developer.symbian.com/main/downloads/papers/static_data/SupportForWriteableStaticDataInDLLsv2.2.pdf**](http://developer.symbian.com/main/downloads/papers/static_data/SupportForWriteableStaticDataInDLLsv2.2.pdf).
- Willee, H. (Dec 2007) 'Symbian OS Notification Services'. Available at [**developer.symbian.com/main/downloads/papers/Notifiers/Symbian_OS_Notification_Services.pdf**](http://developer.symbian.com/main/downloads/papers/Notifiers/Symbian_OS_Notification_Services.pdf).

Index

With thanks to Terry Halliday for creating this index.

- A-GPS 61
- ABI 43
- abstraction benefits,
 - software design 18,
 - 32–4, 94–103,
 - 334–45, 385–95
- actions, services 165–232
- active objects 30, 89, 151
- Active Objects pattern
 - 133–47
 - see also* event...;
 - RunL()
- cancellations 138,
 - 169–70
- Client–Server pattern 146,
 - 147, 182–210
- debugging 142–3, 148
- Event Mixin pattern 146
- long-running active
 - objects 161, 162,
 - 188–9, 306, 328–9
- other patterns and 38, 79,
 - 89, 96, 102, 103,
 - 105, 111, 112, 113,
 - 122, 128, 149, 151,
 - 157, 169, 182–210,
 - 292, 296–7, 306,
 - 308
- panics 30, 140, 159–61,
 - 354–5
- self-completing active
 - objects 162, 329
- servers 146
- single use pattern 146
- stray signals 140, 142–3
- Window Server 146, 352
- active scheduler, concepts
 - 134–47
- ActivityComplete()
 - 197–210
- Adapter pattern 331,
 - 372–84, 395
- Add() 135–47
- AddDeallocCallback()
 - 75–8
- Adopt() 315–29
- after-market application
 - starter 306
- Agenda application 344
- AllFiles 237–8, 240–51
- ALLOC 31
- Allocate 55–62, 65–72,
 - 76–85
- allocation/de-allocation
 - decisions, resource
 - lifetimes 50–85
- AOs *see* active objects
- APIs 19–31, 133–4,
 - 178–81, 207,
 - 234–85
- error-handling strategies
 - 19–31
- multiplexing protocol
 - 19–20
- security issues 234–85
- APPARC 339–45
- AppDllUid 341–5
- AppendL() 296–308
- Application Architecture
 - 339–45
- AppUi 336–45
- ARM... 397–400
 - see also* CPU
- arrays 15, 30, 46–7, 393–4
- ASSERT 24–31, 33–4,
 - 154–5, 224–5, 321,
 - 355–71
- _ASSERT_ALWAYS 24–31
- _ASSERT_COMPILE 24–31
- _ASSERT_DEBUG 24–31,
 - 159–61, 392–3
- asserts
 - see also* panics
 - code 23–4

- asserts (*continued*)
 - concepts 20–31, 33–4, 154–5, 224–5, 321, 355–71, 392–3
 - definition 20–1
 - examples 21
 - execution times 23–7
 - reduced impact 23–4
 - types 20–1
- Asynchronous Controller
 - pattern 132, 147, 148–63, 188–210, 306, 308
- asynchronous event mixins, Event Mixin pattern 102–3
- asynchronous operations 38, 102–3, 105–13, 132, 133–47, 148–63, 166–70, 183–210
 - see also* Active Objects pattern
- atomicity problems, multithreading 132
- Attach() 116–29
- Authenticate() 244–51
- automatic lazy resource proxy 83–4
- AV stream 19–31, 83, 311–12, 324–7
- AVDTP 19–31, 83, 311–12, 324–7
- background activities, Episodes pattern 288, 289–308
- backup server, Publish and Subscribe pattern 127
- batteries
 - see also* power supply constraints 5, 6–7, 49–50, 87–92, 213, 287–8, 310–11
- black boxes *see* encapsulation
- Bluetooth 19–31, 64, 87–9, 261, 269–71, 311–12, 324–7, 383–4
- Bridge pattern 395
- Buckle pattern 252–9
 - concepts 238–9, 248–9, 252–9, 260–1, 284–5, 384
- Cradle pattern 284–5
 - other patterns and 173, 181, 206, 238–9, 248–9, 260–1, 376, 384
- Quarantine pattern 272
- bugs 15–47
 - see also* defects; error...
- Bulk Transaction API 208
- C++ 2, 11, 34–5, 38–9, 143, 351–2, 373, 379–80, 386
- C (heap-allocated) classes 2, 56–8, 67–70, 77–8, 312–29, 389
- cache usage 208, 287, 329
- CActive 91, 122–9, 135–47, 152–63, 169–70, 185–210, 229–30, 299–308
 - see also* Active Objects pattern
- CActiveScheduler 135–47, 152–63
- CActivity 200–10
- CAF 284
- CAkn... 340–5
- Calendar Databases 171–2, 176–8, 204
- camera hardware, Immortal pattern 61–2
- Cancel() 123–4, 135–47, 152–63, 293–308
- CancelActivity() 185–210
- CancelRequest-Complete() 168–70
- CancelService-Request() 168–70
- CancelStateChange() 215–32
- CApaApplication 341–5
- capabilities 174–9, 206, 233–85
- CAPABILITY 178, 256–8, 266–72, 280
- CApaDocument 341–5
- CApaSystemControl... 261, 268–71
- Carbide 344
- case 155–6, 159–61, 194–6, 321–2
- CAsyncCallBack 162, 329
- CAsynchronous-Controller 152–63
- CAsynchronousEpisode 293–308
- category UUIDs 116–29
- CAvdtpInbound-Signalling-Message 324–7
- CBody 387–95
- CBW packets 161
- CClient 135–47, 152–63, 293–308
- CClientActive 185–210
- CCoeAppUi 230–1
- CCoeControl 337–45
- CCoeEnv 352–71
- CCommsDatabase 374, 378–83
- CController 215–32
- CCoordinator.. 215–32
- CDataPress 313–29
- CDialer 143–5
- CDmDomain 228–9
- CDmMmsAdapter 381–3
- CEikApplication 340–5
- CEikAppUi 38–9, 141–2, 340–5
- CEikDocument 340–5
- Central Repository 374
- CEventConsumer 98–102
- CEventGenerator 97–102
- CHandle 389–95

- Cheshire Cat Idiom
 - 385, 387
 - see also* Handle-Body pattern
- child active objects 209
- CHTTPSession 392-3
- CIdle 71, 162
- CImage... 394
- classes
 - see also* C...; M...; R...; T...
 - concepts 2, 11, 13, 21, 56-9, 176, 221-2, 331, 334-45, 346-71, 388-9
 - conventions 13, 94
- CLazyAllocator()
 - 68-70
- CLazyDeallocator()
 - 79-85
- CleanClosePushL()
 - 45-7, 360-71
- cleanup 42-3, 269-71
- CleanupStack 139-47, 296-308, 383
- Client-Server pattern
 - 182-210
 - Active Objects pattern 146, 147, 182-210
 - communication styles 205-6
 - concepts 169-70, 179, 182-210, 226, 232, 247-8, 251, 279-80, 282-5, 353, 365-9, 371, 380-3
 - extensibility 206
 - other patterns and 26-7, 47, 70, 105, 111-13, 115, 129, 133-5, 146, 147, 169-70, 179, 181, 226, 232, 247-8, 251, 279-80, 282-5, 353, 365-9, 371, 380-3
 - performance issues 207-9
 - relative server location 204-5
 - request types 205-6
 - security issues 206
 - server lifetime 205
 - server state changes 205-6, 214-32
 - Singleton pattern 353, 365-9, 371
- Client-Thread Service
 - pattern 171-81
 - compiled code 179-80
 - concepts 169, 184, 259, 380-3
- client.exe 173-81
- clients, services 165-232
- Close() 44-5, 58-9, 122-3
- CMDBSession 379-83
- CMmsAccounts 382-3
- CMmsSettings 381-3
- CMsvEntry 212
- CMyActive 135-47
- CMyAppView 342-5
- CMyAsyncController
 - 153-63
- CMyDocument 341-5
- CMyServer 185-210
- CMyServerSession...
 - 185-210
- CObexServer 101-2
- CObject 39
- CObserver 200-10
- code
 - asserts 23-4
 - conventions 13, 94
 - impact analysis of
 - recurring consequences 400-1
 - RAM 6, 175, 202-3, 248-9, 397-401
 - sample downloads 13-14
 - XIP 6, 27, 397-401
- COggPlay... 291-2, 303-5
- cohesive services 166
- combined
 - acknowledgements and re-registration, Coordinator pattern 232
- CommDB component
 - 374-84
- Comms-Infras *see* Communications Infrastructure
- CommsDat 46-7
- communication channels
 - event-driven chains 89-92
 - resource lifetimes 49-85
- communication styles, Client-Server pattern 205-6
- Communications Infrastructure 34, 43-5, 70, 150-1, 258-9, 305-6, 311-12
- compiled code, Client-Thread Service pattern 179-80
- complete mediation security principle 235
- Complete-Installation 248
- CONE 352, 367-9
- Connect... 21-2, 158-61, 185-210, 248
- constraints on mobile devices 4-6, 12-13, 49-50, 87-92, 131-2, 165-6, 171, 201-3, 212-13, 287-8, 310-11, 347-8, 397-401
- ConstructL() 13, 39-41, 56-62, 66-72, 81-5, 99-102, 123-4, 153-63, 192-210, 291-2, 295-308, 317-29, 355-71, 390
- constructors 13, 39-41, 55-62, 66-72, 79-85, 99-102, 123-4, 136-47, 153-63, 192-210, 291-2, 295-308,

- constructors (*continued*)
 - 317–29, 341–3,
 - 355–71, 386–7, 390
- Contacts Database 149–50,
 - 171–2, 176–8, 204,
 - 208, 336–7
- context switches 143,
 - 202–9, 398,
 - 399–401
- Continue-
 - Installation() 248
- Control Environment 339–45
- Control Panel 268–71
- ControlEnv 368–9
- Controller module,
 - Model–View–Controller pattern 334–45
- conventions 13, 94
- Cookie 206
- cooperative multitasking
 - see also* Active Objects pattern
 - Asynchronous Controller pattern 132, 147, 148–63, 188–210, 306, 308
 - concepts 12, 131–63
- Coordinator pattern 211–32
 - concepts 170, 308
 - multiple processes coordinator 226
 - other patterns and 47, 115, 129, 170, 211–32, 308
 - save notifications 231
 - single-threaded coordinator 215, 219–25
 - view switching 230–1
- CPM 64, 70
- CPolicyServer 279
- CPUs 4–5, 7–8, 49–50, 59, 87–92, 131–2, 302, 306, 310, 323, 391, 397–401
- ARM... 397–400
- constraints 4–5, 7–8, 49–50, 87–92, 131–2, 287–8, 310–11, 397–401
- event-driven chains 89–90
- power-sink components 87–92
- CQik... 340–5
- Cradle pattern 273–85
 - Buckle pattern 284–5
 - concepts 202, 239, 259, 264, 272
- CRawData 313–29
- Create() 56–62, 262–72, 274–85, 337–45, 355–71
- CreateAppUiL() 341–5
- CreateDocumentL() 341–5
- CreateImplementationsL() 255–6
- CRequiredAsyncEpisode 296–308
- CRequiredSyncEpisode 296–308
- CResource 67–70, 78–85
- CResourceOwner 57–62
- CResourceProvider 67–70, 79–85
- CResponder... 215–32
- critical sections 132
- CSaveNotifier() 231
- CSerializer 293–308
- CServer2 192–210
- CSession2 185–210, 282–3
- CSignallingChannel 324–7
- CSink 313–29
- CSmldataProvider 283
- CSmldataSyncUsage 282–3
- CSmldmAdapter 380–3
- CSmldpSession 282–3
- CsocketConnector 158–61
- CSource 313–29
- CStencil 313–29
- CSW packets 161
- CSynchronousEpisode 293–308
- CTelephony 134, 143–5
- CTimer 81–2
- CTS *see* Client-Thread Service pattern
- cts.dll 172–81
- CVersitParser 177–8
- data caching, Client–Server pattern 208
- data cages 183–4, 206, 234–85
- data packets, Data Press pattern 309–29
- data parsing 309–29
- Data Press pattern 309–29
 - concepts 288
 - discover command 324–7
 - error-handling strategies 327–8
 - long-running processes 328–9
- DBMS 171–2, 179, 374–84
- DCLP 359–71
- Deallocate() 55–62, 65–72, 76–85
- de-allocation decisions, resource lifetimes 50–85
- _DEBUGGER 31
- debugging, concepts 17–18, 20–31, 133, 142–3, 148, 157, 249, 392–3
- _DECLARE_TEST 24–31
- decoupled services, Client–Server pattern 202
- defects
 - see also* error...; faults
 - concepts 15–31, 133
 - costs 17–18, 133, 142–3
 - tracking-down difficulties 18

- defense in depth strategy, security issues 233
- Define() 116–29
- Delete() 116–29
- denial-of-service attacks 27, 280, 283
- description template, design patterns 10–12
- descriptors 2, 30, 312–13
- design patterns
 - see also* individual pattern names
 - concepts 1–14, 397–401
 - definition 3, 10
 - historical background 2–4
 - impact analysis of
 - recurring consequences 397–401
 - important considerations 3–4
 - Symbian OS 4–14
 - template 10–12
 - uses 3–4, 10, 397–401
- destructors 55–62, 66–72, 78–85, 124, 350–1, 369–71
- development effort, software designs 8–10, 17–31, 233–85, 394–5
- Device Management (DM) 284, 380–3
- Device Provisioning 284
- device-manufacturer-approved capabilities 236–51
- Digital Rights Management (DRM) 5–6, 20, 238–9, 240–51
- directories, data cages 183–4, 206, 234–85
- discover command, Data Press pattern 324–7
- disk space, resource lifetimes 49–85
- DisplayDialog 248
- DLLs 100, 135, 143, 172–81, 198–210, 233–4, 237–8, 241, 248–9, 252–9, 273–85, 346–71
- DNS 151, 157–61
- DoActivity() 185–210
- DoCancel() 81–2, 123–4, 135–47, 152–63, 185–210, 301–8
- document-centric applications 334–45
- domain errors 15–16, 21–2, 33
 - see also* error...
- DoSomething() 360–3, 389–95
- DoStuff() 57–62
- DoTask() 293–308
- Drawing() 337–45
- DRM files, security issues 5–6, 20, 238–9, 240–51
- dynamic configuration, Episodes pattern 306–7
- dynamically selected CTS 179, 181
- e32base.h 13
- e32def.h 24
- E32Main 139–47, 192–210
- e32std.h 38
- ECom service 179, 253–9, 264–5, 274–85, 376–8, 381–4, 394
- economy of mechanism security principle 235
- EControl... 268–71
- ECriticalBoot 228–30
- EDisconnect 191
- efile.exe 203–4
- efsrv.dll 203–4
- Eikon, Event Mixin pattern 102
- eiksrvs.exe 257–8
- EInvalid... 25–31, 328
- emails 290
- embedded systems 2–3, 4
- emulator 347–9
- encapsulation (black boxes) definition 9
 - software designs 9–10, 18, 114, 121–9, 134–5, 146, 148–63, 166, 292, 336–45, 385–95
- endianess concept 314
- Enhanced 911 – Wireless Service specification (USA) 54
- ENonCriticalBoot 228–30
- EOsServices... 228–30
- ephemeral resources 49–50
 - see also* resource...
- Episodes pattern
 - after-market application starter 306
 - concepts 288, 289–308
 - long-running serializer 306
 - parallelizing episodes 307–8
 - protocol plug-ins 305–6
 - self-determining background episode 306
 - serializer 292, 293–308
- EPOC32 35, 100–1, 127
 - see also* Symbian OS
- EPOCALLOWDLLDATA 354–5
- EPriorityIdle 300–8
- error-handling strategies concepts 12, 15–47, 62, 67–8, 72, 133, 167–70
- Data Press pattern 327–8
- Escalate Errors pattern 16, 31, 32–47, 67–8, 72,

- error-handling strategies
 - (*continued*)
 - 77–8, 96, 101, 102,
 - 121, 129, 140, 155,
 - 184–210, 218–19,
 - 232, 296, 299–300,
 - 320–1, 329, 354–5
- Event Mixin pattern 96
- Fail Fast pattern 16–31,
 - 32, 42, 47, 62, 140,
 - 155, 195–6, 392–3
- leave mechanism 32–47,
 - 97–8, 136, 140,
 - 220–1, 266–7,
 - 296–9, 359–71
- panics 17–31, 33–4, 140,
 - 159–61, 195–6,
 - 354–5, 392–3
- errors
 - see also* KErr...
 - costs 17–18, 27–8, 133,
 - 142–3, 157
 - types 15–16, 21–2
- Escalate Errors pattern
 - 32–47
 - other patterns and 16, 31,
 - 67–8, 72, 77–8, 96,
 - 101, 102, 121, 129,
 - 140, 155, 184–210,
 - 218–19, 232, 296,
 - 299–300, 320–1,
 - 329, 354–5
- Ethernet 34
- EU Universal Service
 - Directive 54
- EUsbMsDriveState...
 - 125–7
- euser.dll 38, 46
- event consumers 88–129
- event generators 88–129
- Event Mixin pattern 93–103
 - see also* mixins
 - Active Objects pattern
 - 146
 - concepts 91, 105, 112,
 - 114, 123, 128, 146,
 - 152–3, 206, 215–32,
 - 293, 313–14, 329,
 - 338
- error-handling strategies
 - 96
- event signals 88–129
- event-driven chains 89–129
- event-driven programming
 - see also* Active Objects
 - pattern; Client–Server
 - pattern; Publish and
 - Subscribe pattern
 - concepts 12, 37–8,
 - 88–129, 131–2,
 - 211–32, 308
 - Coordinator pattern 47,
 - 115, 129, 170,
 - 211–32, 308
 - definitions 88
 - Event Mixin pattern 91,
 - 93–103, 105, 112,
 - 114, 123, 128, 146,
 - 152–3, 206, 215–32,
 - 293, 313–14, 329,
 - 338
 - Request Completion
 - pattern 91, 104–13,
 - 129, 134–5, 146,
 - 147, 165, 167–70,
 - 184–210
- events, definitions 88
- Evictor pattern 85
- EWaitingFor-
 - Connection 151,
 - 158–61
- EWaitingForDNSLookup
 - 151, 158–61
- exceptions *see*
 - error-handling
 - strategies
- execution times 7–8, 13,
 - 23–7, 42, 50, 82,
 - 87–92, 287–329
 - see also* optimization
 - issues
- EXEs
 - security issues 233–85
 - writable static data
 - 348–71
- expected unexpected errors,
 - concepts 21–2
- EXPORT_C 178, 199–200,
 - 203–4, 357, 364,
 - 389–90, 392–3
- extensibility property of
 - services 166, 206
- external asserts
 - see also* asserts
 - concepts 20–2
- EZLib 179
- Façade pattern 16, 18, 384,
 - 395
 - Fail Fast pattern 17–31
 - see also* error...
 - other patterns and 16, 32,
 - 42, 47, 62, 140, 155,
 - 195–6, 392–3
 - faults
 - see also* defects; error...
 - concepts 15–47, 133
 - Feature Manager 306–7
 - field caching, Data Press
 - pattern 329
 - FIFO *see* First-In-First-Out
 - File Server 115–29,
 - 149–50, 183–4,
 - 203–4, 207, 235
 - fire-and-forget 260–72, 308
 - firewalls 18
 - First-In-First-Out (FIFO) 135
 - flash memory 7, 115,
 - 397–401
 - Flyweight pattern 329
 - Font Bitmap Server 208
 - fonts, Lazy Allocation
 - pattern 70
 - Forum Nokia 13–14, 345
 - Freetype 384
 - FSM 149–63
 - function tables, state
 - machines 150
 - functions, conventions 13,
 - 94
 - GAVDP 19–31
 - Get () 116–29, 366–71
 - GetValueL () 185–210

- GIF files 70
- GPS 261
- GUI variants
 - see also* MOAP; S60; UIQ; UIs
 - concepts 13, 331, 332–45, 348–71
- Handle–Body pattern
 - 385–95
 - concepts 331, 384
 - rapid development
 - benefits 394–5
- HandleCommmandL()
 - 342–5
- HandleCompletionL()
 - 137–47
- HandleError() 38–9, 141–2
- HandlePacketTypeL()
 - 313–29
- HandleStateChange–Error() 225–32
- hardware constraints 4–6, 12–13, 49–50, 87–92, 131–2, 201–3, 287–8, 310–11, 397–401
- HeaderField 313–29
- headers, data packets
 - 309–29
- heap 6–7, 397–401
 - see also* RAM
- Highlander *see* Singleton pattern
- hot event signals 88
- HTTP 150–1, 384, 386–7, 391–4
- ICL 70
- Idle Object 71
- iEikonEnv 13
- image converters
 - Handle–Body pattern
 - 394
 - Lazy Allocation pattern
 - 70
- Immortal pattern 53–62
 - see also* resource lifetimes
 - camera hardware 61–2
 - concepts 51, 63–5, 69, 72, 74–5, 82–5, 313, 322–3, 329, 346–7, 388
 - file servers 62
- impact analysis of recurring consequences
 - 397–401
- IMPORT_C 178, 364, 389–95
- information sources
 - 13–14
- infrared 383–4
- INI files 70
- InitializeResource–IfNeededL()
 - 68–9
- InitSingleton() 363–5
- Install 135–47, 248
- Instance 348–71
- internal asserts 20–2
 - see also* asserts
- interprocessor
 - communications (IPC) 26–7, 182–232, 246–7, 274–85, 366–71
 - see also* Client–Server pattern; Coordinator pattern; Publish and Subscribe pattern
- interrupts 88–129
 - see also* event signals
- IP 34, 64, 158–61, 328
- IPC *see* interprocessor communications
- Isolate Highly Trusted Code
 - see* Secure Agent
- IssueAsyncRequest()
 - 136–47
- Java 345, 384
- JPEG files 70
- kernel 15, 112–13, 114–29, 183–4, 203, 234–5, 398–401
- KERN-EXEC 3 panic 354
- KErrAlreadyExists
 - 187–8
- KErrArgument 97–8
- KErrCancel 156–63, 167–70, 190–210
- KErrNoMemory 15–16, 35, 39
- KErrNone 33, 108–9, 141–2, 167–70, 190–210, 217–32, 277–8, 299, 320–1, 365–71
- KErrNotFound 15–16, 117
- KErrNotReady 151, 160–1
- KErrNotSupported 347, 384
- KErrPermissionDenied
 - 255–9, 277
- keyboard constraints 6
- keys, properties 116–29
- Kill 264
- KRequestPending
 - 107–13, 142–7
- KUIdSystemCategory
 - 117
- L2CAP 311–12
- LAN 205
- latency issues
 - Client–Server pattern 202
 - Data Press pattern 309–29
- layering considerations, Coordinator pattern
 - 211–32
- Lazy Allocation pattern
 - 62–72
 - see also* resource lifetimes
 - concepts 51–2, 84, 346–7, 350, 371, 393–4
- Lazy De-allocation pattern
 - 73–85

- Lazy De-allocation pattern
 (continued)
 see also resource lifetimes
 concepts 52, 70, 72, 191,
 205, 209
- LBS 54, 60–1
- LDD 249
- Leasing pattern 85
- least privilege security
 principle 235
- leave mechanism 32–47,
 97–8, 136, 140,
 220–1, 266–7,
 296–9, 359–71
- LIBRARY 177–8, 245–6,
 256–7
- ListImplementationsL()
 278–85
- LIT_SECURITY_
 POLICY... 117,
 120–1, 126–7
- LoadAllProtocols()
 213–32
- LocalServices 261,
 270–1, 282–3
- Location 261
- Logical Device Driver (LDD)
 249
- long-running active objects
 161, 162, 188–9,
 306, 328–9
- long-running processes,
 Data Press pattern
 328–9
- long-running serializer,
 Episodes pattern 306
- loose coupling 387–95

- M (abstract interface) classes
 2, 94–103
 see also mixins
- McDeregister-
 Responder()
 215–32
- McRegister-
 ResponderL()
 215–32
- macros
 asserts 24–31
 trap mechanism 39–47
- McStateChange-
 Complete()
 215–32
- mailinit.exe 271–2
- maintainability issues
 optimization trade-offs
 287, 292
 software designs 9–10,
 17, 133, 142–3,
 148–9, 171, 287–8,
 292, 343
- malware 233–85
 see also security...
- manufacturers 5–6, 50,
 236–51
- many publishers, single
 subscriber, Publish
 and Subscribe pattern
 127–8
- marshaling process 182–3
- mass storage, Asynchronous
 Controller pattern
 161–2
- Master–Slave pattern 209
- Mayfly pattern 51, 64, 74
 see also resource lifetimes
- MCoeView 230–1
- MCoordinator 215–32
- MecEpisodeComplete
 293–308
- media player 290–2, 302–5
- Mediator 232
- MeExecute... 293–308
- Memento 209
- MemEventHas-
 OccurredL()
 94–103
- memory 4–5, 6–7, 15–16,
 331, 347–8,
 397–401
 see also RAM; ROM
- MEpisode 293–308
- MEpisodeCompletion
 293–308
- message queues 279–80,
 366–9
- Messaging application 344
- messaging engine, Event
 Mixin pattern 102–3
- messaging initialization,
 Quarantine pattern
 271–2
- MEventMixin 94–103
- MFC 344–5
- microkernel architecture
 183–4
- Microsoft Foundation
 Classes (MVC) 344–5
- MicTaskComplete
 293–308
- middleware 288, 309–29
- mixins 94–103
 see also Event Mixin
 pattern
- MMP files 174, 178, 181,
 245–9, 256–9,
 266–72, 280, 354–5
- MMS 381–3
- MMU 209
- MOAP 2, 332
- MOBexServerNotify 100–2
- mobile devices
 constraints 4–6, 12–13,
 49–50, 87–92,
 131–2, 165–6, 171,
 201–3, 213, 287–8,
 310–11, 347–8,
 397–401
 PC comparisons 18,
 49–50
 reliability expectations 5,
 8, 16
 security expectations 5–6,
 8, 13, 17
 stakeholders 5–6
 upgrades 5
- Mock Objects 100
- Model–View–Controller
 pattern (MVC) 13,
 331, 332–45
- Monostate 371
- MOODS 163
- moving memory model
 397–401
- MPacketEvents 313–29

- MpeInvalidPacket 295–308, 353–71, 386, 392–3
- MpePacketTypeEvent 313–29
- MrCancelState-Change() 215–32
- MResponder 215–32
- MrHandleState-Change() 215–32
- MSA 115–16, 125–9
- MscStateChange-Complete() 215–32
- MStateChanger 215–32
- MTaskCompletion 293–308
- MTaskEventMixin 152–63
- MtcTaskComplete 296–308
- multiple controllers, Coordinator pattern 231
- multiple inheritance 94–5
- multiple memory model 397–401
- multiple publishing
 - frequencies, Publish and Subscribe pattern 128–9
- multiplexing protocol 19–20
- multitasking 12, 131–63, 292
 - see also* cooperative...
- multithreading, concepts 12, 131–2, 142–3, 157, 350–1, 398
- mutexes 132, 358–63
- MVC *see* Model–View–Controller pattern
- network operators, security expectations 5–6
- NetworkServices 270–1, 282–3
- New... 39, 99–102, 123–4, 137–47, 153–63, 295–308, 353–71, 386, 392–3
- NewApplication() 337–45
- NewData() 313–29
- NewSession() 185–210
- NextStep 197–210
- Nokia 13–14, 344–5
 - see also* S60
- non-fatal errors 32–47
- non-pre-emptive process scheduling 131–2
 - see also* cooperative multitasking
- Notification Server 253, 258
- NotifyCurrentLayer() 223–32
- NotifyEpisode-Complete() 299–308
- NotifyLayer() 218–32
- NotifyNextLayer() 223–32
- NotifyTaskComplete() 295–308
- OBEX 93–4, 100–1, 384
- objects
 - Model–View–Controller pattern 13, 331, 332–45
 - Singleton pattern 13, 176, 221–2, 331, 346–71
- Observe() 185–210
- Observer *see* Event Mixin pattern
- OggPlay 290–2, 302–5
- OMA 382
- Open... 43–5, 58–9, 121, 392–3
- open design security principle 235
- open operating systems 5
- open-access secure agent 250
- OpenGL 384
- optimization issues
 - see also* execution times; performance... complexity problems 287–8, 292 concepts 13, 42, 82, 287–329 Data Press pattern 288, 309–29 Episodes pattern 288, 289–308 maintainability trade-offs 287, 292 premature optimization dangers 287 simple tactics 287–8 out-of-bound array accesses 15, 30 overloaded operators 39 overview of the book 12–13 ownable resources 49–85 *see also* resource...
- P&S *see* Publish and Subscribe pattern
- packets, Data Press pattern 309–29
- PAN 205
- Panic() 17–31, 159–61, 195–6, 392–3
- panics 17–31, 33–4, 140, 159–61, 195–6, 354–5, 392–3
 - see also* asserts
- parallelizing episodes 307–8
- parsing 309–29
- patched software 16, 18
- pattern theory 3
 - see also* design patterns
- patterns *see* design patterns
- Patterns of Events 147
- payloads, data packets 309–29
- PC comparisons, mobile devices 18, 49–50

- peek (get) requests,
 - Client–Server pattern 205–6
- Peer-to-Peer 209
- pens 332–3
- performance issues 207–9
 - see also optimization . . .
- Permanent servers 205
- Phone application 336–7, 344
- PIM applications 172, 204
- pInstance 348–71
- Pipes 209
- PIPS 380–3
- platform security see security . . .
- plug-ins 64, 237–85, 305–6, 331, 372–84
- poke (post) requests,
 - Client–Server pattern 205–6
- polling, constraints 87–8, 212–13
- pooled allocation 84–5
- Pop() 296–308, 383
- PopAndDestroy() 139–47
- POSIX 374, 380
- power supply constraints 5, 6–7, 49–50, 87–129, 213, 310–11
 - see also batteries
- power-sink components 87–92
- PowerMgmt 268–71
- pre-emptive process
 - scheduling 131–63
- PrepareResourceL() 75–85
- priority inversion 145
- private 234
- privilege leakage security
 - principle 235
- Process . . . 313–29
- processes
 - context switches 399–401
 - creation times 399–400
 - impact analysis of
 - recurring consequences 398–401
 - RAM usage 398–400
 - Singleton pattern 13, 176, 221–2, 331, 346–71
 - trust zones 233–4, 237–85
- Programming by Contract
 - see Fail Fast pattern
- programming errors 15–47
 - see also error . . . ; faults
- properties
 - Publish and Subscribe pattern 115–29
 - services 166–7
- Protected System 251
- protocol modules, Buckle pattern 258–9
- protocol plug-ins, Episodes pattern 305
- ProtServ 280–3
- providing services 12
- Proxy pattern 18, 184–5, 209, 216–17, 232, 251, 285, 366–9, 384
- Publish and Subscribe (P&S) pattern 114–29
 - many publishers, single subscriber 127–8
 - multiple publishing frequencies 128–9
 - other patterns and 61, 91, 92, 103, 111–13, 206, 246, 279–80, 366–7
 - shared definitions 120–1, 125–7
- pure virtual functions, M (abstract interface)
 - classes 94–5
- PushL() 139–47
- PXT plug-in 248–9
- Quarantine pattern 260–72
 - Buckle pattern 272
 - other patterns and 239, 257, 259, 273–9, 281, 285
- plug-ins 266–72
- R (resource) classes 2, 58–9, 67–70, 77–8, 389
- radio units, constraints 87
- RAM 4–5, 6–7, 49–50, 55–6, 60–2, 64, 74, 83, 120, 132–4, 142–3, 148, 157, 165–6, 171–2, 175, 202–3, 247, 268, 276, 281, 309, 310, 321–3, 347, 397–401
- code 6, 175, 202–3, 247, 397–401
- constraints 4–5, 6–7, 49–50, 132–4, 142–3, 148, 157, 165–6, 171, 202–3, 287–8, 310–11, 347, 397–401
- impact analysis of
 - recurring consequences 397–401
- predictable usage 7
- processes 398–400
- reduction motivations 6–7
- threads 132–4, 142–3, 148, 157, 397–8
- RArray 46–7, 393–4
- RAScliSession 212
- RawData 312–29
- RDmDomain 228–9
- RDriveStateChanged–Publisher 126–7
- Read() 21–2
- ReadDeviceData() 237–8, 261, 268–71
- ReadUserData() 282–3
- REComSession 234–5, 255–6, 278–85, 382–3

- References design pattern
 - template 12, 31, 47
- registering multiple events,
 - Event Mixin pattern 102
- RegisterUserData() 303
- RegisterView() 337–45
- relative server location,
 - Client–Server pattern 204–5
- Release() 45–6
- ReleaseResourceL() 75–85
- reliability expectations,
 - mobile devices 5, 8, 16
- reliable event signals 91, 114–29
- Rendezvous() 263–72
- Rendezvous service
 - 110–13, 263–4
- Request 376–84, 393–4
- Request Completion pattern 104–13
 - other patterns and 91, 129, 134–5, 146, 147, 165, 167–70, 184–210
- RequestComplete
 - 107–13, 138–47, 162, 244–51, 277–85
- RequestSession-
 - HeadersL() 393–4
- Reset() 313–29
- resource 234
- resource clients 52–85
- resource lifetimes
 - allocation/de-allocation decisions 50–85
 - concepts 12, 49–85
 - definitions 49
 - Immortal pattern 51, 53–62, 63–5, 69, 72, 74–5, 82–5, 313, 322–3, 329, 346–7, 388
 - Lazy Allocation pattern 51–2, 62–72, 84, 346–7, 350, 371, 393–4
 - Lazy De-allocation pattern 52, 70, 72, 73–85, 191, 205, 209
 - Mayfly pattern 51, 64, 74
 - resource providers, concepts 52–85
 - resources, concepts 49–85, 201–2
 - response timeouts,
 - Coordinator pattern 231
 - ReStart() 185–210
 - restricted system
 - capabilities, concepts 236–8
 - RFComm Protocol
 - Implementation 328
 - RFile 33
 - RFs 33, 203–4
 - RHandleBase 116–29, 387
 - RHostResolver 159–61
 - RHTTPSession 150–1, 391–3
 - RHTTPTransaction 150–1
 - RLibrary 234–5, 259
 - RMBufChain 312–29
 - RMessage2 185–210
 - RMyClientSession 185–210
 - RNotifier 253–9
 - ROM 83, 176–7, 246
 - RPointerArray 295–308
 - RProcess 26–7, 234–5, 262–72, 274–85, 387
 - RProperty 61, 115–29, 366–71
 - RProperty::Define() 61
 - RPropertyPublisher 121–9
 - RResourceProvider 58–9
 - RSC files 264
 - RSessionBase 185–210, 279–85
 - see also Client–Server pattern
 - RSmlCSDDataProvider 282–3
 - RSocket 34–5, 43–5, 59–62
 - RSocketServ 45–7, 213–32
 - RThread... 26–7, 106–13, 360–71, 387
 - RunError() 38, 81–2, 135–47, 152–63, 300–8
 - RunL() 38, 81–2, 91, 102–3, 123–4, 135–47, 151–63, 185–210, 299–308, 329
 - RunPluginL() 266–7
 - RunThreadL() 139–47
 - S60 2, 13–14, 269, 271, 332–45, 348–54
 - background 332–3, 348–54
 - Model–View–Controller pattern 332–45
 - sample code downloads 13–14
 - SAP 29
 - save notifications,
 - Coordinator pattern 231
 - SCSI 161
 - SDKs 13–14, 228–9, 344
 - SDL see Symbian Developer Library
 - secondary storage,
 - constraints 6–7, 49–50
 - Secure Agent
 - concepts 238–9, 240–51, 262, 285
 - extensions 249–51

- Secure Agent (*continued*)
 - IPC 246
 - open-access secure agent 250
- Secure ID (SID) 117, 181, 183–4, 233–85
- security credentials 233–85
 - see also* capabilities; SID; VID
- security issues
 - Buckle pattern 173, 181, 206, 238–9, 248–9, 252–9, 260–1, 272, 273–80, 284–5, 376, 384
 - capabilities 174–9, 206, 233–85
 - Client–Server pattern 206
 - concepts 5–6, 8, 13, 17, 20, 27, 117, 120–1, 126–7, 206, 233–85
 - Cradle pattern 202, 239, 257, 259, 264, 272, 273–85
 - data cages 183–4, 206, 234–85
 - defense in depth strategy 233
 - DLLs 233–4, 237–8, 241, 248–9, 252–9, 273–85
 - DRM files 5–6, 20, 238–9, 240–51
 - mobile devices 5–6, 8
 - principles 233–8
 - process trust zones 233–4, 237–85
 - Quarantine pattern 239, 257, 259, 260–72, 273–9, 281, 285
 - Secure Agent 238–9, 240–51, 262, 285
 - types of risk 6, 27
- security privileges 285–8
 - see also* capabilities
- self-completing active objects 162, 329
- self-determining background episode 306
- semaphores 105–13, 132
- SendResponse() 313–29
- separation of concerns, service properties 166
- SEPs 324–7
- serializer, Episodes pattern 292, 293–308
- server lifetime, Client–Server pattern 205
- server state changes, Client–Server pattern 205–6, 214–32
- servers
 - see also* file...; window...
 - Active Objects pattern 146
 - Client–Server pattern 26–7, 47, 70, 105, 111–13, 115, 129, 133–5, 146, 147, 169–70, 179, 181, 182–210, 226, 232, 247–8, 251, 279–80, 353, 365–9, 371, 380–3
 - service providers, services 165–232
 - service requests, concepts 165–232
 - ServiceError() 195–210
 - ServiceL() 185–210
 - ServiceRequest() 277–85
- services
 - asynchronous actions of a service 166–70, 183
 - Client–Server pattern 26–7, 47, 70, 105, 111–13, 115, 129, 133–5, 146, 147, 169–70, 179, 181, 182–210, 226, 232, 279–80
 - Client-Thread Service
 - pattern 169, 171–81, 184, 259, 380–3
 - concepts 165–232
 - Coordinator pattern 47, 115, 129, 170
 - definition 165
 - properties 166–7
 - synchronous actions of a service 166–70, 183
 - sessions 165–232
 - Set() 116–29, 263–72
 - SetActive() 137–47, 153–63
 - SetConfigurationL() 30
 - SetTheCoe() 367–9
 - shared data and atomicity
 - problems, multithreading 132
 - shared definitions, Publish and Subscribe pattern 120–1, 125–7
 - Shared Library *see* Client-Thread Service pattern
 - shared memory, Client–Server pattern 208
 - ShutdownL() 41–2
 - SID 117, 181, 183–4, 233–85
 - SignalStateChange() 215–32
 - signed applications 236–85
 - single use pattern, Active Objects pattern 146
 - Singleton pattern 13, 176, 221–2, 331, 346–71
 - Single Thread
 - implementation 352–6, 366–71
 - Client–Server pattern 353, 365–9, 371
 - concepts 331, 346–71
 - destructors 350–1, 369–71
 - hazards 350–1

- synopsis of solutions 352–3, 367–8
- system-wide Singleton pattern
 - implementation 353, 365–71
- thread-managed Singleton pattern within a process
 - implementation 353, 363–5, 366–71
- thread-safe ... within a process
 - implementation 357–63, 366–71
- TLS singleton in a single thread
 - implementation 353, 356–7, 363–5, 366–71
- Singleton class 348–71
- SingletonPtr 363–5
- SIS files 242–51
- SMP 131
- SMS messages 128, 344
- sockets 34–5, 150–1, 158–61, 204
- software designs
 - see also* error ...
 - constraint factors 4–10, 50, 87–92, 176, 287–8, 310–11
 - development effort 8–10, 17–31, 233–85, 394–5
 - major forces 4–10, 16
 - patched software 16, 18
- software errors *see* error ...
- software installer (SWI) 241–51
- Solution design pattern
 - template
 - concepts 20–30, 35–43
 - definition 11–12
- Sony Ericsson 332–3, 345
 - see also* UIQ
- sources of information 13–14
- SpecificRequest 376–84
- splash screens 290–2, 302–5
- SQL 172, 374–5, 379–80
- stack
 - see also* RAM
 - concepts 6–7, 19–20, 36–8, 47, 397–401
- Staged Initialization *see* Episodes pattern
- Staged System Transition *see* Coordinator pattern
- stakeholders, mobile devices 5–6
- Start() 135–47, 185–210
- StartServer() 199–210
- state machines,
 - Asynchronous Controller pattern 148–63
- state patterns 150, 163
- state transitions checking 21, 29–30
- State-Aware Components
 - see* Coordinator pattern
- stateful services 165–6
- stateless services 165–6
- static allocation *see* Immortal pattern
- STATICLIBRARY 181
- Status Reporting 227, 231–2
- stencil, Data Press pattern 313–29
- Stop 45–6, 135–47
- stray signals 140, 142–3
- sub-tasks 148–9
 - see also* Asynchronous Controller pattern
- Subscribe() 116–29
 - see also* Publish and Subscribe pattern
- SWI 241–51
- switch 151–3, 155–6, 157, 159–60, 194–6, 321–2
- switch statements and an enumerated state, state machines 150
- Symbian Debug Security Server 249
- Symbian Developer Library (SDL) 13–14, 47, 58, 129, 138–9, 178–9, 181, 192, 210, 256, 259, 263, 266, 280, 344
- Symbian Foundation platform 2
- Symbian OS
 - Active Objects pattern 89, 96, 102–3, 105, 111–13, 122–3, 128, 132–47
 - Adapter pattern 13, 331, 372–84, 395
 - background 1–47
 - Client–Server pattern 26–7, 47, 70, 105, 111–13, 115, 129, 133–5, 146, 147, 169–70, 179, 181, 182–210, 226, 232, 247–8, 251, 279–80, 353, 365–9, 371, 380–3
 - Communications Infrastructure 150–1, 258–9, 305–6, 311–12
 - constraint factors 4–10, 12–13, 49–50, 87–92, 131–2, 165–6, 176, 201–2, 287–8, 310–11, 397–401
 - conventions 13, 94
 - design considerations 4–10, 16
 - design patterns 4–14
 - File Server 115–29

- Symbian OS (*continued*)
 - GAUDP/AVDTP 19–31, 83, 311–12, 324–7
 - Handle–Body pattern 331, 384, 385–95
 - HTTP 150–1, 384, 386–7, 391–4
 - important software forces 6–10
 - Model–View–Controller pattern 13, 331, 332–45
 - patched software 18
 - RFComm Protocol
 - Implementation 328
 - security issues 233–85
 - Singleton pattern 13, 176, 221–2, 331, 346–71
 - Telephony subsystem 134, 143–5, 204
 - two-phase construction
 - critique 386–7
 - USB 104–13, 115–29, 384
 - v8 261, 347, 351
 - v9.x releases 2, 43, 70, 83, 172, 233, 249, 268, 306, 331, 337, 347–9, 351–8, 374, 401
 - well-known patterns 13, 331–95
- Symbian Remote Control Framework 30
- Symbian Signed 237–8
- synchronization problems, multithreading 132
- synchronous actions of a service 166–70, 183–210
- SynchronousService–Request 244–51
- SyncML 161–2, 274, 282–3, 384
- sys 234
- system capabilities, concepts 236–8
- system errors
 - see also* error ...
 - concepts 15–16, 21–2, 33
 - system-wide Singleton pattern
 - implementation 353, 365–71
 - SYSTEMINCLUDE 177–8
 - T (data-type) classes 2, 388
 - TApaAppcapability 268–71
 - target interface, Adapter pattern 331, 372–84
 - TARGETTYPE 178–81, 245–6, 256–7
 - tasks
 - Episodes pattern 288, 289–308
 - multitasking 12, 131–63, 292
 - sub-tasks 148–9
 - TAVStreamState 30
 - TCA 384
 - TCB 106–7, 183–4, 234–85
 - TCE 235–85
 - TConnectState 158–61
 - TCP/IP 34, 64, 158–61, 328
 - TechView UI layer process 268
 - Telephony subsystem 134, 143–5, 204
 - template overview, design patterns 10–12
 - testing 9–10, 23–31, 133, 142–3, 148–9, 157, 332–45
 - _TEST_INVARIANT 24–31
 - TFindFile 264–5
 - TFixedArray 57–8
 - TheCoe 367–9
 - third-party developers 348
 - thread request semaphore 105–13
 - thread-managed Singleton 353, 363–5, 366–71
 - threads 12, 13, 17–47, 87–129, 131–2, 142–3, 157, 169, 171–81, 184, 221–2, 259, 331, 346–71, 380–3, 397–401
 - see also* Active Objects pattern
 - Client-Thread Service pattern 169, 171–81, 184, 259, 380–3
 - context switches 143, 202–9, 398, 399–401
 - creation times 398
 - event-driven programming 88–129
 - impact analysis of
 - recurring consequences 397–401
 - multithreading 12, 131–2, 142–3, 157, 350–1, 398
 - panics 17–31, 140
 - power-sink components 87–92
 - RAM usage 132–4, 142–3, 148, 157, 397–8
 - Singleton pattern 13, 176, 221–2, 331, 346–71
 - time zone server, Publish and Subscribe pattern 127
 - Tls 356–7, 363–5, 367–71
 - TLS singleton 353, 356–7, 363–5, 366–71
 - TOCTOU 255–6
 - TPckgBuf 121
 - TPriQueLink 135–47
 - transaction batching 208
 - TransferPayload 313–29
 - transient servers 70, 83, 205, 209
 - TRAP 39–47, 266–72
 - trap mechanism, concepts 32–47, 266–72, 360–3
 - TRAPD 40–7, 360–3
 - TRAP_IGNORE 40–7

- TRequestStatus 44–5, 91, 102–3, 104–13, 117–29, 135–47, 152–63, 188–210, 265–72, 291–2
- trust zones
 - see also* security issues
 - processes 233–4, 237–85
- Trusted Computing Base (TCB) 106–7, 183–4, 234–85
- Trusted Computing Environment (TCE) 235–85
- Trusted Extension Component 206
- Trusted Proxy 251
- TrustedUI 247–8
- TState 152–63
- two-phase construction, critique 386–7
- typedef 121

- UDP 34–5, 43–6
- UIDs 116–29, 233, 264–5, 341–5
- Uikon 339–45
- UIQ 2, 13–14, 141–2, 269, 271, 332–45, 348–54
 - background 332–3, 348–54
 - Developer Community 13–14
- UIQ2 348
- UIQ3 333, 337, 342, 348–9
- UIs 332–45, 348
 - see also* GUI...
- un-marshaling process 182–3
- Unicode text 116
- unit-test code 23
 - see also* testing
- Universal Service Directive (EU) 54
- upgrades, mobile devices 5
- URLs, conventions 13
- usability property of services 166, 207
- USA Enhanced 911 – Wireless Service specification 54
- USB 104–13, 115–29, 384
- Use 55–62, 65–72, 76–85, 388
- user capabilities, concepts 236–8
- user interfaces, constraints 6, 45–7
- User::Leave... 38–9, 40–1, 97–8, 122, 154–63, 266–71, 359–71
- User::Panic() 24–31
- User::WaitForAnyRequest() 106–13
- User::WaitForNRequest() 106–13
- User::WaitForRequest() 44–5, 106–13, 138–47, 199–210, 270–1, 291–2
- ValidateInputs() 244–51
- ValidatePacketTypeL() 313–29

- variables, conventions 13
- Vendor ID (VID) 117, 181, 233–85
- Versit 177–8
- VID *see* Vendor ID
- View module, Model–View–Controller pattern 334–45
- view switching, Coordinator pattern 230–1
- viruses 18

- WaitForRequest() 44–5, 106–13, 138–47, 199–210, 270–1, 291–2
- WAN 205
- warm communication relationships 205
- warm event signals 88
- web 74, 205
- well-known patterns, concepts 13, 331–95
- wiki page 1, 14
- Window Server, Active Objects pattern 146, 352
- writable static data (WSD) 173, 176, 331, 346–71
- WriteDeviceData 117, 261, 268–71
- WriteUserData 282–3
- WSD *see* writable static data
- WSERV 146, 352
- WSP 384

- XIP 6, 27, 397–401

- Y-Browser 258

Error-Handling Strategies

Fail Fast (page 17) Improve the maintainability and reliability of your software by calling `Panic()` to deal with programming errors as soon as they are detected.

Escalate Errors (page 32) Enhance your error handling by using the Symbian OS Leave mechanism to escalate the responsibility of handling an error up to the point which has enough context to resolve the error.

Resource Lifetimes

Immortal (page 53) Allocate a resource as soon as possible so that, when you come to use it later, it is guaranteed to be both available and quickly accessible.

Lazy Allocation (page 63) Allocate resources, just in time, when they are first needed, to improve how they are shared across the system.

Lazy De-allocation (page 73) Delay the de-allocation of a resource so that an almost immediate re-use of the resource does not incur a performance penalty.

Event-Driven Programming

Event Mixin (page 93) Save power and decouple your components by defining an interface through which event signals are sent to another component in the same thread.

Request Completion (page 104) Use `TRequestStatus` as the basis for event signals sent across thread and process boundaries.

Publish and Subscribe (page 114) Securely broadcast changes to other threads or processes listening for such notifications as a way of saving power and decoupling components.

Cooperative Multitasking

Active Objects (page 133) Enable efficient multitasking within a single thread by using the Symbian OS active object framework.

Asynchronous Controller (page 148) Encapsulate a finite state machine within an active object to efficiently control a modest set of related asynchronous sub-tasks.

Providing Services

Client-Thread Service (page 171) Enable the re-use of code by rapidly developing a service that executes independently within the thread of each client.

Client–Server (page 182) Synchronize and securely police access to a shared resource or service from multiple client processes by using the Symbian OS client–server framework.

Coordinator (page 211) Co-ordinate state changes across multiple interdependent components by using a staged notification procedure that maximizes flexibility.

Security

Secure Agent (page 240) Minimize security risks by separating out security-critical code into a separate process from non-critical code.

Buckle (page 252) Load DLL plug-ins, which match your level of trust, into your own process to increase the flexibility of your architecture without compromising security.

Quarantine (page 260) Load plug-ins as separate processes, operating at different levels of trust to your own, to increase the flexibility of your architecture without compromising security.

Cradle (page 273) Host DLL plug-ins in separate processes, operating at different levels of trust to your own, to securely increase the flexibility of your architecture whilst maintaining communication with each plug-in.

Optimizing Execution Time

Episodes (page 289) Give the appearance of executing more swiftly by delaying all non-essential operations until after the main task has been completed.

Data Press (page 309) Quickly process an incoming stream of data packets, without allocating any further memory, by using a long-lived object to interpret each packet in turn.

Mapping Well-Known Patterns onto Symbian OS

Model–View–Controller (page 332) Allow an interactive application to be easily extended, ported and tested by dividing the responsibility for managing, displaying and manipulating data between three cooperating classes.

Singleton (page 346) Ensure a class has only one instance and provide a global point of access to it.

Adapter (page 372) Transform calls on one interface into calls on another, incompatible, interface without changing existing components.

Handle–Body (page 385) Decouple an interface from its implementation so that the two can vary independently.