The Symbian OS Architecture Sourcebook

Design and Evolution of a Mobile Phone OS

By
Ben Morris

Reviewed by
Chris Davies, Warren Day, Martin de Jode, Roy Hayun, Simon Higginson, Mark Jacobs, Andrew Langstaff, David Mery, Matthew O'Donnell, Kal Patel, Dominic Pinkman, Alan Robinson, Matthew Reynolds, Mark Shackman, Jo Stichbury, Jan van Bergen

Symbian Press

Head of Symbian Press
Freddie Gjertsen

Managing Editor
Satu McNabb



John Wiley & Sons, Ltd

SAMSUNG 1023

The Symbian OS Architecture Sourcebook

The Symbian OS Architecture Sourcebook

Design and Evolution of a Mobile Phone OS

By

Ben Morris

Reviewed by

Chris Davies, Warren Day, Martin de Jode, Roy Hayun, Simon Higginson, Mark Jacobs, Andrew Langstaff, David Mery, Matthew O'Donnell, Kal Patel, Dominic Pinkman, Alan Robinson, Matthew Reynolds, Mark Shackman, Jo Stichbury, Jan van Bergen

Symbian Press

Head of Symbian Press

Freddie Gjertsen

Managing Editor

Satu McNabb



John Wiley & Sons, Ltd

Copyright © 2007 Symbian Software, Ltd
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.wileyeurope.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.

Anniversary Logo Design: Richard J. Pacifico

Library of Congress Cataloging-in-Publication Data

Morris, Ben, 1958-
The Symbian OS architecture sourcebook : design and evolution of a mobile phone OS / by Ben Morris.
p. cm.
Includes bibliographical references.
ISBN-13: 978-0-470-01846-0
ISBN-10: 0-470-01846-1
1. Operating systems (Computers) 2. Symbian OS (Computer file) I. Title.
QA76.76.O63M6835 2007
005.4'32 – dc22

2006103533

British Library Cataloguing in Publication Data

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

ISBN: 978-0-470-01846-0

Typeset in 10/12pt 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.

To Philippa, with love.

Contents

About this Author xiii

Acknowledgements xv

Glossary of Terms xvii

Introduction xix

Part 1: The Background to Symbian OS

1 Why Phones Are Different 3
- 1.1 The Origins of Mobile Phones 3
- 1.2 From 2G to 3G 5
- 1.3 Mobile Phone Evolution 6
- 1.4 Technology and Soft Effects 7
- 1.5 Disruption and Complexity 9
- 1.6 The Thing About Mobile Phones 10

2 The History and Prehistory of Symbian OS 15
- 2.1 The State of the Art 15
- 2.2 In the Beginning 17
- 2.3 The Prehistory of Psion 20
- 2.4 The Beginnings of Symbian OS 22
- 2.5 The Mobile Opportunity 26
- 2.6 Background to the First Licensee Projects 27
- 2.7 Device Families 31
- 2.8 Operating System Influences 37

3 Introduction to the Architecture of Symbian OS **45**
3.1 Design Goals and Architecture 45
3.2 Basic Design Patterns of Symbian OS 49
3.3 Why Architecture Matters 49
3.4 Symbian OS Layer by Layer 52
3.5 The Key Design Patterns 56
3.6 The Application Perspective 65
3.7 Symbian OS Idioms 71
3.8 Platform Security from Symbian OS v9 83

4 Introduction to Object Orientation **87**
4.1 Background 87
4.2 The Big Attraction 88
4.3 The Origins of Object Orientation 90
4.4 The Key Ideas of Object Orientation 92
4.5 The Languages of Object Orientation 100

Part 2: The Layered Architecture View

5 The Symbian OS Layered Model **111**
5.1 Introduction 111
5.2 Basic Concepts 111
5.3 Layer-by-Layer Summary of the Symbian OS v9.3 Model 117
5.4 What the Model Does Not Show 119
5.5 History 119

6 The UI Framework Layer **121**
6.1 Introduction 121
6.2 Purpose 122
6.3 Design Goals 123
6.4 Overview 123
6.5 Architecture 124
6.6 A Short History of the UI Architecture 128
6.7 Component Collections 129

7 The Application Services Layer **133**
7.1 Introduction 133
7.2 Purpose 134
7.3 Design Goals 134
7.4 Overview 135
7.5 Legacy Application Engines 137
7.6 Architecture 137
7.7 Component Collections 149

8 The OS Services Layer 165
8.1 Introduction 165
8.2 Purpose 166
8.3 Design Goals 168
8.4 Overview 170
8.5 Architecture 171
8.6 Generic OS Services Block 171
8.7 Multimedia and Graphics Services Block 177
8.8 Connectivity Services Block 192

9 The Comms Services Block 199
9.1 Introduction 199
9.2 Purpose 201
9.3 Design Goals 204
9.4 Overview 206
9.5 Architecture 206
9.6 Comms Framework 210
9.7 Telephony Services 220
9.8 Networking Services 230
9.9 Short-link Services 245

10 The Base Services Layer 255
10.1 Introduction 255
10.2 Purpose 255
10.3 Design Goals 256
10.4 Overview 257
10.5 Architecture 258
10.6 Component Collections 270

11 The Kernel Services and Hardware Interface Layer 279
11.1 Introduction 279
11.2 Purpose 280
11.3 Design Goals 281
11.4 Overview 283
11.5 EKA1 and EKA2 283
11.6 Singleton Component Collections 284
11.7 Kernel Architecture Block 285
11.8 Kernel Architecture Component Collections 295

12 The Java ME Subsystem 301
12.1 Introduction 301
12.2 Requirements of the Java ME Subsystem 302
12.3 Design Goals for the Java ME Subsystem 302
12.4 Evolution of Java on Symbian OS 303

12.5 Architecture 306
12.6 Component Collections 311

13 Notes on the Evolution of Symbian OS 319
13.1 The State of the Art 319
13.2 Summary of Symbian OS v6 Releases 319
13.3 Summary of Symbian OS v7 Releases 321
13.4 Summary of Symbian OS v8 Releases 324
13.5 Summary of Symbian OS v9 Releases 326

Part 3: Design Case Studies

14 The Use of Object-oriented Design in Symbian OS 333
14.1 Introduction 333
14.2 Pioneering the Object Approach in Psion 334
14.3 A Thoroughly Object-oriented Operating System 353

15 Just Add Phone 367
15.1 Introduction 367
15.2 Anatomy of a Phone 367
15.3 The Phone Operating System 368
15.4 Telephony 378
15.5 Messaging: It's Different on a Phone 386

16 One Size Does Not Fit All: The Radical User Interface Solution 397
16.1 Introduction 397
16.2 Background to the Eikon GUI 402
16.3 Eikon Design Point 404
16.4 The Device Family Strategy 410
16.5 Quartz 416
16.6 Pearl 417
16.7 Nightingale 418
16.8 How to Develop a World-class GUI 420
16.9 Symbian OS User Interface Architecture 425
16.10 Future Directions 426

17 System Evolution and Renewal 429
17.1 Introduction 429
17.2 Design Lifetime 430
17.3 Renewal in Symbian OS 434
17.4 Evolution in the Kernel 436
17.5 Telephony Evolution 440
17.6 Sound and Vision Evolution 443

17.7 Defining the Skin 444
17.8 Moving Towards Standard C++ 446

18 Creative Zoo or Software Factory? 453
18.1 Introduction 453
18.2 The Software Problem 453
18.3 Too Many Dragons 455
18.4 Software Development Approaches 456
18.5 What Making Software Is Really About 459

Appendix A: Symbian OS Component Reference 475

Appendix B: Interviewee Biographies 573

References 579

Index 583

About the Author

Ben Morris 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***

Acknowledgements

Some people told me it would be hard to write this book in and around my real job in the System Management Group at Symbian and a few promised me that it would be impossible. They were all right, of course, although none of them tried to stop me.

Many thanks to Wiley and Symbian Press therefore for their patience as I've stretched deadlines. Thanks to Fredrik Josephson for saying 'yes' to my starting the book as a 10% task and for turning a blind eye when it grew beyond that; and to Geert Bollen for being (almost) tolerant when he inherited the problem. Thanks to Freddie Gjertsen of Symbian Press for getting me to the end and to Phil Northam for his part in making it happen in the first place.

My biggest thanks, though, are due to those who took the time to talk to me, agreed to my using a recording device and let me use their words. They are: Geert Bollen, Martin Budden, Andy Cloke, Charles Davies, Bob Dewolf, Morgan Henry, Ian Hutton, Peter Jackson, Keith de Mendonca, Will Palmer, Howard Price, Murray Read, Martin Tasker, Andrew Thoelke and David Wood. I have done my best to make sure they are happy with the use to which I have put their words.

I am also very grateful to my technical reviewers from across the company (and, in a few cases, from outside it): Jan van Bergen, Chris Davies, Warren Day, Roy Hayun, Simon Higginson, Mark Jacobs, Martin de Jode, Andrew Langstaff, David Mery, Matthew O'Donnell, Kal Patel, Dominic Pinkman, Matt Reynolds, Alan Robinson, Mark Shackman, Phil Spencer, and Jo Stichbury. Jeff Lewis provided a final review from a commercial perspective.

Any errors which remain are mine, of course.

A special thanks to Jawad Arshad for his help in constructing the reference material in Appendix A, and for his careful review of what

I did with it, and to Bob Rosenberg for his great work on the System Model graphics (which is present in the book in the form of the color pull-out). Way back when, Martin Hardman was my original collaborator on early versions of the System Model, and I would like to acknowledge his contribution

Finally, my family have put up with this book for longer than was promised. Philippa, Nat, Jake and Henrietta – thanks.

Glossary of Terms

ABI	Application binary interface
ADT	Abstract data type
BAL	Bearer Abstraction Layer
BIO	Bearer-independent object
CDMA	Code Division Multiple Access
DFRD	Device family reference design
DRM	Digital rights management
DSP	Digital Signal Processor
EDGE	Enhanced Data Service for GSM Evolution
ETSI	European Telecommunications Standards Institute
FOMA	Freedom of Mobile Access
GPRS	General Packet Radio Service
IPC	Interprocess communication
MOAP	Mobile Application Platform
MTM	Message type module
MVC	Model–view–controller
OBEX	IrDA Object Exchange
OMA	Open Mobile Alliance
OTA	Over the air
PAN	Personal Area Networking
PIM	Personal information manager
PLP	Psion Link Protocol
QoS	Quality of Service
RTOS	Real-time operating system
RTP	Real-time transport protocol
SIP	Session initiation protocol

SMIL	Synchronized Multimedia Integration Language
UART	Universal Asynchronous Transmitter/Receiver
UMTS	Universal Mobile Telecommunications System
VoIP	Voice over IP
VPN	Virtual Private Network
WAP	Wireless Application Protocol
WDP	Wireless Datagram Protocol
XIP	Execute in place

Introduction

This book is part description, part reference, part case study and part history. My goal in writing it has been to try to make Symbian OS more accessible to a wider audience than has been catered for to date. I hope there is nothing dumbed-down about this book, but at the same time I have tried to make it accessible to those who are interested, but not expert, in the topics it covers, as well as useful to a more hands-on developer audience.

As Symbian OS becomes more mainstream – a volume product and not just a niche one – I hope this book will serve as a primer for the curious and a way in to a deeper understanding of what Symbian OS is, where it came from and why it is currently riding high.

Certainly there is material here which is useful to Symbian OS developers – both seasoned and novice – and which has previously been hard to find. However, this book takes a different approach to that of most Symbian Press books; it is not so much a 'how to' book as a 'what and why' book (and to some extent also a 'who and when' book).

Part 1 is a Symbian OS primer, a rapid introduction that sketches the background of the mobile telephony market, traces the emergence of Symbian OS and Symbian the company, conducts a rapid tour of the architecture of Symbian OS, and provides a refresher – or introduction – to the key ideas of object orientation (OO) in software.

Part 2 begins the more detailed exploration of the architecture of Symbian OS, following the Symbian OS System Model layering to provide a complete, high-level, architectural description of Symbian OS.

Part 3 returns to the historical approach of the primer chapters, and presents five case studies, each exploring some aspect of Symbian OS, or of its history and evolution, in depth. Drawing on the insights – and the

recollections – of those who were involved, these studies trace and try to understand the forces that have shaped the operating system.

Appendix A contains a component by component reference, ordered alphabetically by component name, which is definitely intended for a developer audience. It also includes a color pull-out of the System Model for the current public release, Symbian OS v9.3.

Who This Book Is For

This book is for anyone who wants to understand Symbian OS better – what Symbian OS is, why it is what it is, and how it got to be that way. If you work with Symbian OS, or intend to, this book is for you. If you want to get under the skin of the OS and understand it more deeply, this book is very definitely for you. This book is for you too if you are interested in the software or mobile phone industries more generally, or in the perennial themes of software development, or are merely curious about how real systems get made and evolve.

A reasonable degree of software technical literacy is assumed, but not so much that the more casual reader should shy away. There are no exercises. And there is no sample code.

How to Use This Book

This book calls itself a *sourcebook* and it is intended to be used both as a primer and as a reference. Its different sections are useful in their different ways as reference material. Both Part 1 and Part 3 are structured as a straight-through read and, I hope, they offer a good starting point from which to come to Symbian OS for the first time. The material in Part 2 is probably deeper than a non-developer audience needs. And while this is not (strictly) a programming book, I hope that Symbian OS developers find its reference material useful, or better.

Telling Stories

Someone else wrote the phrase before I did: ''In every great software product is a great story'' [McCarthy 1995]. I think it's true. So while this book is aimed at a technically aware audience, it is not addressed exclusively to an audience of programmers. I hope programmers and, more generally, software developers, designers and architects will find it useful, especially those coming new to the OS and trying to understand it. But I hope it will be just as useful to academics and students, marketeers, technical decision makers and managers seeking to evaluate

and understand Symbian OS, and indeed anyone else who is broadly in the business of software or phones or who is just interested in such things, and who is encountering Symbian OS (or its close competitors) for the first time and needs to understand it. Speaking personally, I have long been something of an operating system junkie; to some extent, therefore, this book attempts to scratch that itch. (You can't work for an operating system company and not have a bit of the operating system junkie in you.)

I hope that understanding the deeper story behind Symbian OS will help those who want to (or have to) work with it to understand it better and more deeply. Above all, I hope it will help them work better with Symbian OS than would be the case without this book.

I have another purpose too. One of the things which appealed to me most in my early days in the company (which became Symbian a few months after I joined) was the degree to which everyone involved in creating the system shared the sense that making software is a visionary activity and that making good software, indeed the best possible software, is as much a moral imperative as a business one. For an activity which likes to count itself as a branch of engineering, the number, and variety, of value words which cropped up in any daily conversation could be surprising. Making software, which is to say making this software in particular, is value-laden. 'Delight', 'elegance', 'trust', 'integrity', 'robustness', 'reliability', 'economy' and 'parsimony' were all among the company buzzwords and very much part of the fabric of the effort, and give a flavor of those times. Above all, to be part of the effort to create Symbian OS was to be part of the revolution, no less. The truly personal, individual, pocketable, always-on, human-scaled device you could trust your data to, and to some extent therefore also your identity, and your heart as well as your head, was not yet the commonplace thing which the mobile phone revolution has made of it. Symbian – the operating system and the company – has played its part, too, in that revolution.

Symbian is currently riding high. Symbian OS has done more than find a niche; it has found (and, indeed, it has founded) a global market and has led that market from its inception. To make that point more concretely, consider this: when I was starting work on this book, I drafted a paragraph about 2005 being a watershed year for Symbian OS, potentially its breakout year. Between then and now, as I write this at the end of 2006, the number of shipped Symbian OS phones has doubled from 50 million to 100 million, and counting.

Way back when, the company was a company of individuals – who could be opiniated, strident and arrogant but could just as quickly switch to humility in the face of a powerful intellectual argument. Inevitably, some of that individuality has been lost with success and growth. I hope that by capturing some of the flavor of those times, that particular flame can be kept burning.

I have been mindful both of commercial and personal confidences and I believe that nothing I have written (or quoted) breaches either. (Any instances of 'Don't quote me!' which appear in the text have been carefully approved.)

I have tried everywhere to observe the mantra 'Tell no lies', which is not always the case in books such as this, and which here and there has not been easy. Let me quote Bjarne Stroustrup as one inspiration for honesty, 'I abhor revisionist history and try to avoid it'.[1] I have done my best to follow that example.

Getting Symbian OS

Anyone, anywhere, can download Symbian OS in a form in which they can learn to program it, work with it, explore it and experiment with it. Anyone can learn to write Symbian OS applications: development kits are free, and easily available, for UIQ and S60 platforms; development tools (GCC and Eclipse) are free; the Symbian Press programming books are widely available; and the possible languages range from OPL (which began life as the Psion Organiser Language and is now an open-source, rapid application development language for phones based on Symbian OS) and Visual Basic (available from AppForge), through Java and Python, to full-on native Symbian OS C++. The range is covered, in other words, for everyone from the hobbyist to the enterprise developer to phone manufacturers and commercial developers.

[1] In [Stroustrup 1994, p2].

Part 1

The Background to Symbian OS

1

Why Phones Are Different

1.1 The Origins of Mobile Phones

The first mobile phone networks evolved from the technologies used in specialist mobile phone radio systems, such as train cab and taxi radios, and the closed networks used by emergency and police services and similar military systems.

The first ever open, public network (i.e., open to subscribing customers rather than restricted to a dedicated group of private users) was the Autoradiopuhelin (ARP, or car radio phone) network in Finland. It was a car-based system, inaugurated in 1971 by the Finnish state telephone company, that peaked at around 35 000 subscribers [Haikio 2002, p. 158].

A more advanced system, the Nordic Mobile Telephone (NMT) network, was opened a decade later in 1981 as a partnership between the Nordic state telecommunications monopolies (of Denmark, Finland, Norway and Sweden), achieving 440 000 subscribers by the mid-1990s, that is, more than a ten-fold increase on ARP [Haikio 2002, p. 158]. Unlike ARP, a car boot was no longer required to house the radio hardware. Ericsson, and later Nokia, were primary suppliers of infrastructure and phones, helping to give both companies an early edge in commercial mobile phone systems.

Elsewhere, Motorola and AT&T competed to introduce mobile phone services in the Americas, with the first Advanced Mobile Phone System (AMPS) network from AT&T going public in 1984. European networks based on an AMPS derivative (Total Access Communication System, TACS) were opened in 1985 in the UK (Vodafone), Italy, Spain and France.[1] Germany had already introduced its own system in 1981. In

[1] See for example the company history at ***www.vodafone.com***.

Japan, a limited car-based mobile phone service was introduced in 1979[2] by NTT, the not-yet privatized telecommunications monopoly, but wider roll-out was held back until 1984. A TACS-derived system was inaugurated in Japan in 1991.

All these systems were cellular-based, analog networks, so-called first-generation (1G) mobile phone networks (ARP is sometimes described as zeroth-generation).

The history of the second-generation (2G) networks begins in 1982 when the Groupe Speciale Mobile (GSM) project was initiated by ETSI, the European telecommunications standards body, to define and standardize a next-generation mobile phone technology,[3] setting 1991 for the inauguration of the first system with a target of 10 million subscribers by 2000. GSM was endorsed by the European Commission in 1984; spectrum agreements followed in 1986; and development began in earnest in 1987. GSM reflected a deliberate social as well as economic goal, that of enabling seamless communications for an increasingly mobile phone world as part of the wider project to create a unified Europe. The politics of deregulation was also an important factor in the emergence of new mobile phone networks as rivals to the traditional monopoly telecommunications providers.[4]

The first GSM call was made, on schedule, in Finland on 1 July 1991, inaugurating the world's first GSM network, Radiolinja. By 1999, the network had achieved three million subscribers, a ten-fold increase on first-generation NMT and a hundred-fold increase on ARP.

GSM rapidly expanded in Europe, with new networks opening in the UK (Vodafone, Cellnet, One2One and Orange), Denmark, Sweden and Holland, followed by Asia, including Hong Kong, Australia and New Zealand. By the mid-1990s, new GSM networks had sprung up globally from the Philippines and Thailand to Iran, Morocco, Latvia and Russia, as well as in the Americas and to a lesser extent the USA, making GSM the dominant global mobile phone network technology.

Through the 1990s, GSM penetration rose from a typical 10% after three years to 50% and then 90% and more in most markets (all of Europe, for example, with the Nordic countries leading the way, but with Italy

---

[2] A useful history appears at ***www2.sims.berkeley.edu/courses/is224/s99/GroupD/project1/paper.1.html***.

[3] For a history of GSM see ***www.gsmworld.com/about/history.shtml***, as well as [Haikio 2002, p. 128].

[4] Political events unfolding between 1988 and 1992, such as the pulling down of the Berlin Wall, German unification and the collapse of the Soviet Union, were also indirectly significant, for example in causing Nokia to refocus on the mobile phone market [Haikio 2002, Chapters 5 and 7].

and the UK not far behind). By the end of the decade, the USA and Japan were atypical, with the USA opting for a different technology (CDMA[5]) and Japan languishing at less than 50% GSM penetration.[6]

1.2 From 2G to 3G

Famously, 3G is the technology that the network operators are most frequently said to have overpaid for, in terms of their spectrum licenses. (Auctions of the 3G spectrum raised hundreds of billions various currencies globally in the first years of the 21st century.)

In the GSM world, 3G means UMTS, the third-generation standard designed as the next step beyond GSM, with a few half-steps defined in between including GPRS, EDGE (see [Wilkinson 2002]), and other '2.5G' technologies. In the CDMA world, 3G means CDMA2000. (In other words the division between the USA and the rest of the world persists from 2G into 3G.)

The significant jump that 3G makes from 2G is to introduce fully packetized mobile phone networks. (GPRS, for example, is a 'halfway' technology that adds packet data to otherwise circuit-switched systems.) The significance of packetization is that it unifies the mobile phone networks, in principle, with IP-based (Internet technology) data networks. Japan has led the field since a large-scale 3G trial in 2001 but, as of the last quarter of 2005, it seems that 3G has arrived 'for the rest of us', with the introduction (finally) of competitively priced 3G networks from the likes of Vodafone and Orange in Europe, opening the way for competition to improve the 3G network offering.

Disappointingly, in terms of services 3G has not yet found a distinct identity. But from the phone and software perspective, the story is rather different. Early problems with the greater power drain compared to GSM, for example, made for clunky phones and poor battery life. Those problems have been solved and 3G phones are now interchangeable with any others. From a software perspective, there are no longer particular issues. Symbian OS has been 3G-ready for several releases. (From a user perspective, of course, 3G is different because it is 'always on'.)

---

[5] CDMA, also known as 'spread spectrum' transmission, was famously co-invented in a previous career by Hedy Lamarr, the Hollywood actress. [Shepard 2002] provides a very approachable survey of telecommunications technologies. [Wilkinson 2002] is an excellent, mobile phone-centric survey.

[6] [Haikio 2002, p. 157] presents figures for mobile phone network penetration for 20 countries between 1991 and 2001.

1.3 Mobile Phone Evolution

Mobile phones for the early analog networks were expensive, almost exclusively car-mounted devices selling to a niche market. Equipment vendors sold direct to customers. Network operators had no retail presence and generated cash flow solely from call revenues. As the analog networks evolved into GSM networks, mobile phones were liberated from the car and the early car phones evolved into personal portable phones and then began to shrink until they fitted, firstly, into briefcases and, finally, into pockets. From around 1994, when GSM started to boom, mobile phones and perhaps even more importantly mobile phone network services began to emerge as potential mass-market products.

The iconic Mobira Cityman, introduced by Nokia in 1986, was the size of a small suitcase and, with its power pack, weighed in at nearly 800 grams [Haikio 2002, p. 69]. By 1990, phones had halved in size and weight and they had halved again by 1994, when the Nokia 2100 was released. It was the first ever mass-market mobile phone and weighed in at 200 grams [Haikio 2002, p. 160]. (It is credited with selling 20 million units, against an initial target of 400 000.)

As it happens, 1998, the year that Symbian was created, saw a temporary market reversal[7] but mobile phone uptake boomed again towards the turn of the millennium.[8]

The PC and mobile phone trend lines crossed in 2000 when mobile phones outsold personal computers globally for the first time[9] (by a factor approaching four: 450 million phones to 120 million PCs). This was also the year in which the first Symbian OS phone shipped, the Ericsson R380, followed in 2001 by the Nokia 9210. Neither were volume successes but both products were seminal. In particular, the Nokia 9210 instantly put Nokia at the top of the sales league for PDAs, ahead of Palm, Compaq and Sharp. (The Communicator was classified by market analysts as a PDA, partly because it had a keyboard, but also partly because Symbian phones really were a new category, and analysts didn't quite know what to do with them.) The death of the PDA, much trumpeted since (and real enough, if Microsoft's Windows CE sales numbers and the demise of Palm OS are indicators), probably dates from that point.[10]

---

[7] Nokia failed to meet sales targets; Motorola issued a profits warning and cut jobs; Philips canceled joint ventures with Lucent; Siemens cut jobs; and Ericsson issued profits warnings.

[8] Mobile phone telephony thus acquires something of a millennial flavor, see [Myerson 2001, p. 7].

[9] Market data for the period can still be found on the websites of market analysis companies such as Canalysis, Gartner, IDC and others, as can the subsequent wider coverage from news sites ranging from the BBC and Reuters to The Register.

[10] In Q3 2005, for example, PDA shipments fell 18% while smartphone shipments rose 75% year on year. See, for example, commentary at The Register, ***www.theregister.co.uk***.

Although in 1997 Nokia shipped just over 20 million mobile phones, in 2001 it shipped 140 million and the trends were broadly similar for other vendors. (Nokia was the clear leader with over 30% of the market in 2001, compared to second placed Motorola with closer to 14%.) Even so, numbers which looked astonishing in 2001 [Myerson 2001] look decidedly tame today. In 2005, global mobile phone sales broke through the barrier of 200 million phones per quarter, with year-end shipments of 810 million, close to 20% and shipments for 2006 rising a further 21%, almost touching the 1 billion mark.[11] Close to 40% of the sales growth in 2005 came from Eastern Europe, Africa and Latin America.

Against these totals, annual sales of smartphones at closer to 50 million in 2005 look small (which is why Symbian has begun to chase the mid-range market). Nonetheless, Symbian OS still leads the market, having doubled its shipments in pretty much every year since the company's creation. Thus, shipments in 2003 more than doubled from 2 million to 6.7 million; in 2004, they doubled again to 14.4 million; and in 2005 they more than doubled again, with almost 34 million Symbian OS phones shipped in the year (see ***www.symbian.com/news/pr/2006/pr20063419.html***).

1.4 Technology and Soft Effects

Almost as astonishing as the raw numbers are the social and technology changes packed into little more than half a decade. The Nokia 7650, introduced in spring 2002, was a breakthrough product. The first camera phone in Europe [Haikio 2002, p. 240] with MMS, email, a color display and a joystick, the Nokia 7650 introduced the Series 60 (now rebranded as S60) user interface and was the first Symbian phone to sell in significant volume. Looking back, it is easy to forget how novel its camera was.

Not even five years on, the mobile phone seems well on the way to subsuming digital photography (the digital camera market began to shrink for the first time in 2005, although arguably that may indicate saturation as much as competition). It is an open question whether mobile phones will do the same to the personal music-player market.[12] Phones seem already to have subsumed PDAs. This is the principle of convergence; on the evidence of the market to date, given the choice between multiple dedicated single function devices or multifunction mobile phone terminals (as mobile phones are increasingly described), the market is choosing the latter.

---

[11] See ***www.gartner.com/it/page.jsp?id=501734***

[12] Apple's Quarter 1 2006 sales numbers, for example, show a decline in iPod sales at the same time as the Nokia 3250 'music player' phone has hit 'triple platinum' (i.e. 1 million units shipped) within a single quarter.

It may not even matter what impact convergence has on existing markets. Broadcast TV, Wi-Fi, and VoIP[13] are queuing up for the role of newest hot mobile phone technology and seem likely to sustain continued growth, with or without markets such as the personal music player. (Digital terrestrial broadcast TV may yet prove to be the 'killer app' for the mobile phone.) What seems certain is that personalization has worked. Whatever the market drivers (and they are not necessarily the same in all markets), person-to-person communications have moved from their Victorian origins in fixed lines anchored to fixed locations, to what used to be the distinctly science-fictional model of ubiquitous mobile phone personal communications (something rather more like the Star Trek model).

Genuine culture shock accompanied the emergence of the mass mobile phone market, with its new habits and behaviors: people chatting into their phones in the street and breezily answering their phones in restaurants and trains, breaking the unwritten rules of public–private spaces and frequently meeting hostility in consequence. Similarly, the rapid rise of a 'texting' culture produced a predictable gap between those who did (typically young users) and those who didn't, with an equally predictable spate of newspaper scare stories. Today, these seem like reports from a world long gone. Looking back at the vision for the future mobile phone information society that Nokia began promoting from around 1999, it is remarkable how much of it has come to pass. The vision is spelled out in detail in [Kivimaki 2001].

Telephony has always had a sociological dimension, ever since the fixed-line phone shrank the world and collapsed time, making two-way communications between remote locations instantaneous. This is even more striking for mobile telephony. Again, it is easy to forget how completely in the UK, for example, the first brick-like mobile phones became the personification of the London 'Big Bang' deregulation of the City, of the Thatcher era and the Lawson boom, every bit as much as red Ferraris. (Local TV news reported at the time that motorists were buying dummy mobile phones, simply to be seen talking into them while waiting at the traffic lights, thus catching some of the Big Bang glow.) Again, the curious notion of the 'car phone' has left its legacy in the name of one of the UK's larger mobile phone retailers, Carphone Warehouse. (Elsewhere in Europe, where the sociology presumably was different, the brand is simply Phone House.)

The mobile phone is an astonishing product phenomenon. Not just the businesses of the phone vendors, but completely new operator businesses too have been built on the back of selling and serving the mobile phone. New business models have been invented from subsidies for phones

[13] Voice Over IP (VoIP) telephony uses non-dedicated IP networks to carry voice telephony traffic. Internet phone services such as Skype are VoIP-based as, increasingly, are discount packages offered by mainstream phone providers.

to pre-payment and the marketing of intangibles such as 'airtime' and 'messages'. Meanwhile some old business models have collapsed under pressure from the cannibalization of neighboring markets including fixed-line telephony.

It is easy to underestimate the depth of these 'soft' effects. The PC brought about several social revolutions: as the visible embodiment of the ubiquitous microprocessor, as the medium for the Internet, including email, and most recently as the medium for the web. Arguably, the mobile phone transformation runs even deeper, because it impacts public and not just private behavior. It has both caused and enabled new social uses (it has changed family relationships, enabled 'remote mothering' [Ling 2004, p. 43] and so on), as well as new patterns of behavior which have rapidly become the norm (it has changed the way much business is done, changed the way people set up meetings, and melted private–public distinctions). The mobile phone 'fits into the folds of everyday life' (L. Fortunati quoted in [Ling 2004, p. 51]) in a way that few other technologies have and the effect has been extremely powerful.

1.5 Disruption and Complexity

A strong theme of this book is that mobile phones are uniquely complex, both as devices and as products, and are therefore uniquely challenging from a software perspective. Of course many things are complex. Rockets are complex and so is the Internet, and so are corporate services, battleships and submarines. But mobile phones outdo them all in the complexity of the package.

Mobile phones are complex packages of multiple software functions (computing, communications and multimedia), hardware technologies (battery and power, radio, displays, optics (lenses), and audio), and fabrication and manufacturing technologies (miniaturization, online customization and localization, global procurement, and sourcing and distribution) which are sold globally in unprecedented volumes. They have moved from a niche market to the mainstream in two decades, with much of that growth in the last five years. They have been technologically, commercially, and socially disruptive.

The typical pattern of a disruptive technology is that it succeeds not by outperforming existing technologies (many disruptive technologies have in fact failed first time around as direct challengers), but by subtly shifting the ground on which it competes. Instead of competing like-for-like, it outperforms the incumbents on shifted ground, in effect skewing the existing market and creating a new, related and overlapping but essentially different market. It removes the ground beneath the old technology not by replacing it directly but by sidelining it, often by moving the market in an unprecedented direction. It is rather like adaptive evolution, in which

an unproductive mutation becomes unexpectedly relevant and therefore successful because of a shift in the external context.[14]

Disruption is part of what makes it so hard to predict the future. WAP failed dismally in one market whereas i-Mode was a runaway success in another, but on the face of it both offer essentially the same service. The missing ingredient for success in the case of WAP was not a technology ingredient but a market or social one. Andrew Seybold says that i-Mode 'is a cultural success – not a wireless success' (quoted in [Funk 2004, p. 13]). While the analysis is probably only half true, it does make the point that the social and cultural dimensions of technologies cannot be ignored.

Arguably, convergence is itself a form of disruption. One reason to believe that the mobile phone will dominate at the expense of laptops, PDAs, digital cameras or dedicated music players, all of which are objectively fitter for a single purpose, is that while these devices may score higher on function (in their niche), they score lower on personalization and value as an accessory. Symbian OS does not itself count as a disruptive technology, but it is a vehicle for the disruptive effects of convergence.[15]

1.6 The Thing About Mobile Phones

Mobile phones are different from other devices for many reasons and most of those reasons make them more complex too.

- Mobile phones are multi-function devices.
- Mobile phone functionality is expanding at an exponential rate.
- Phone-related technologies are evolving at an exponential rate.
- Mobile phones are enmeshed in a complex and still evolving business model.
- Mobile phones are highly personal consumer devices (even when someone else pays for them).

In a word, the mobile phone difference is 'complexity' and the trend towards complexity appears to be growing at an exponential pace.

---

[14] Disruption is a widely discussed (and fashionable) concept, first identified by Christensen [1997] as innovative change for which the market is the trigger point (see [Tidd 2005, p. 29]. [Funk 2004, p. 4] has a simple definition. [Davila 2006] defines it neatly as 'semi-radical technology innovation'.

[15] Symbian OS is, of course, itself at risk from the disruptive business model offered by Linux.

Mobile Phone Hardware and Software

Baseband (radio 'modem') hardware is complex. In effect, the baseband hardware is a complete package in its own right, consisting of CPU, data bus, dedicated memory, memory controller, digital signal processors (DSPs), radio hardware, and so on.

The baseband software stack is complex too. Mobile phone protocols are complex and require real-time systems to support their signaling timing tolerances. Real-time support cannot be faked. A real-time operating system is required at the bottom of the stack to manage the hardware and support the layers of software protocols all the way up to the phone-signaling stack.

Treating the phone as a black box encapsulated by a communications protocol simplifies the software problem but has drawbacks in terms of both speed and capability. The power and speed requirements of the phone's hardware cannot be ignored.

Mobile Phone Applications

A typical Symbian OS phone has a complete application suite: phone book application, email and messaging clients, jotter, clock and alarm applications, connection and network setup utilities (not to mention web browser, camera support and photo album applications, video clip player and editor, and music player).

The application layer requires a full function graphical user interface (GUI) framework to support it, from widget set to full application lifecycle. Most (and probably all) of the expected applications also demand fairly deep system support from the operating system.

While Symbian OS staked its initial claim at the high end of the market, partly on the strength of its application support, the downward push towards the mass-market volumes of mid-range phones does not mean it has to do less. There are persuasive arguments that the mid-range is not defined by functional breadth (the range of available applications) so much as by functional depth (the size of the mailbox, the number of fields in a contact, and so on). Equally, critical factors such as performance are typically more demanding in the mid-range, where users have higher expectations that things 'just work' and lower tolerance for failure.

Convergence and Commoditization

Phone functionality is extending in every direction. Two-camera phones are becoming commonplace, true optical cameras have arrived (with Zeiss lenses, for example), as have phones with boom-box stereo speakers, a gigabyte of RAM and a built-in global positioning system (GPS).[16]

[16] Siemens and Mitac for example have both announced GPS-enabled GSM phones.

Device convergence is not a hypothesis, it is the reality. As discussed above, mobile phones have cannibalized the PDA market, appear to have eroded the digital camera market, and threaten other markets including the personal music-player market.

At the same time, new technologies and advances in existing technologies continue to be relentlessly absorbed into and commoditized by the mobile phone market. For example, Wi-Fi is causing the connection model for mobile phones to be reinvented, with hot-spot connectivity offering alternative network options. Meanwhile advances in storage media, from flash drive densities to micro hard-drives, challenge the use-case assumptions for mobile phones. From being the equivalent of snapshot cameras, they have become full video-recording devices; with internal memories of several gigabytes, they now compete with dedicated music players.

While the PC market, for example, has been essentially mature for a decade and now exhibits little more exciting than consolidation, the mobile phone market continues to be transformed by convergence and commoditization.

Services

Possibly the biggest difference between phones and other mobile devices is the integration of uniquely complex technology with uniquely complex business models. Phone services are almost as important in the product offering as immediate phone functionality.

Everything about the mobile-phone-network business model is complex, from spectrum licenses to roaming, to network subsidies for phones, to packetization of data and the interaction with legacy technology models, be they fixed-line telephony or radio and TV broadcasting or the Internet.

This complexity has its impact on the software in mobile phones, whether it is the requirement to support custom network services, to enable customized applications, or to be invisible beneath the top-line branding of networks and vendors (which leads, for example, to the demand to support custom user interfaces).

Open Platforms

Symbian OS sets out to be an open application platform, in other words a platform for which anyone can write and sell (or share, or simply give away), installable software, whether end-user applications and utilities or service and feature extensions.

Symbian therefore must provide the development tools and support (tool chains, support programs, compatibility guarantees and documentation, including books) needed by external developers to understand and

use the system, and to design and write stable and secure applications to run over various releases of the operating system and on various phone models, including phones from different vendors.

Open platforms are easy to promise and hard to deliver. Success can present acute problems of scaling. Thus, for example, while vendors were bringing to market only one or two models per year, it was possible for third parties to test their applications on all available phones. Those days are long gone, with the biggest Symbian licensees sometimes bringing out a dozen or more Symbian-based models in a single quarter. Managing the success of the platform means managing compatibility better; adopting and adhering to open standards including tool and language standards (standard C++, the ARM EABI, and so on); producing more and better documentation; providing more developer services such as the Symbian Signed program; the list could go on. In turn, these things can only be achieved by creating a healthy ecosystem around the platform to increase the overall pool of available resources and maximize the community contribution.

User Expectations

Users expect and demand rock-solid stability and performance from their phones; desktop computer performance standards are not acceptable.

At the same time, users are fickle, tending either to be infinitely happy or infinitely unhappy.[17] When they are infinitely unhappy, they return the phone. However, it is not always easy to understand precisely what triggers happiness or unhappiness (the trigger often seems removed from ordinary measures of good, bad and defective behavior). Desktop PC users seem more likely to be either *infinitesimally* happy (the machine has not crashed) or unhappy (it crashed but they did not lose much data).

The conclusion is that phones really are different from other systems and they are complex.

[17] Thanks to Phil McKerracher for this idea.

2

The History and Prehistory of Symbian OS

2.1 The State of the Art

Symbian OS reached market for the first time towards the end of 2000, with the release of the Ericsson R380 mobile phone in November and the announcement almost immediately afterwards of the Nokia 9210 Communicator, which came to market in June 2001. Both phones were based on versions of what had previously been known as Psion's EPOC operating system. The final EPOC release was EPOC32 Release 5 (strictly speaking, the final version was the full Unicode build, designated ER5u). The first release of Symbian OS was therefore designated v6.0.

Since then, well over a hundred phone models later (the 100th model[1] shipped in early Q2, 2006) and with more than 100 million (and rising) cumulative unit sales, Symbian OS has undergone continuous evolution to keep pace with the rapidly changing technology in the market it targets: communications-enabled mobile terminals including, of course, mobile phones.

The latest release of Symbian OS is v9. In v9, and its precursor v8, dozens of new APIs offer access to services and technologies which in many cases simply did not exist when Symbian OS first launched.[2] Bluetooth support was one of the earliest additions (v6); Wi-Fi is one of the most recent (v9). Telephony support, meanwhile, has evolved from basic GSM and GPRS (in v6) to include EDGE (v7), CDMA (v8) and 3G (v8). Networking support including IPSec has been integral from

[1] The Nokia 3250 (also, as it happens, the first Symbian OS v9.1 phone to market) was the 100th model, reaching the shops in April 2006, soon followed by the Sony Ericsson P990, also based on v9.1.

[2] To name just the three most obvious examples, Java ME, Bluetooth and 3G networks did not exist when Symbian OS was first launched.

the beginning, evolving to a dual IPv4/v6 stack in v7 and enabling full Internet browsing on Symbian phones, with recent additions including support for VPN clients. New multimedia APIs (v8) support the high data rates required for two-way streaming and high definition interactive TV (DVB-H). The graphics system supports vector graphics (OpenGL ES in v8), with direct screen access and double resolution displays. The new platform security model (v9) enables the platform to remain open, but safe, with a signing service to support trusted application download. The list goes on.

The foundation for these latest services is the new real-time kernel (available in v8.1b and from v9), supporting the multiple fast interrupts needed for high data throughput, the latest generation of ARM processor architectures (ARMv6 is supported in v9) and single core phone designs.

The latest Symbian OS phones are full multimedia devices, including multimegapixel cameras with integrated flash and optical zoom, support for hot-swappable media cards up to 2 GB, MP4 (video) and MP3 (audio) players (supporting WMA and AAC too), 24-bit color (16.7 million colors), not to mention Wi-Fi, and Universal Plug and Play (UPnP, which enables remote control of compatible PCs, audio systems and TVs from a Symbian phone).

Having achieved its first '1 million phones shipped' year in 2002 (2.1 million Symbian OS phones were shipped that year, compared with 0.5 million the year before), Symbian OS achieved 1 million phones shipped in one quarter in Q1 2003, and 1 million phones shipped in one month in December 2003. Volumes have continued to rise steadily since then, and Symbian OS passed the 100 million phones shipped milestone in Q4 2006. Those numbers translate into close to 70% of the high-end, or smartphone, market according to independent sources.[3]

At the same time, competition has probably never been greater. In 2006, Linux phones are shipping in substantial numbers in Japan and China. Microsoft launched the latest version of its mobile phone platform, Windows Mobile 5, in late 2005 and phones based on it are now shipping.[4] Qualcomm has signed up European networks for the first time to support its (previously CDMA-only) Brew platform. And while the future of the Java-based SavaJe platform is uncertain, 'all-Java' phones remain a possibility.[5]

But, at the time of writing, the biggest volume of phones (i.e. across the whole market) are based on none of these platforms at all and remain the mid- to low-end phones based on vendors' own proprietary operating systems. In 2006, Symbian set its sights on addressing this market and its

[3] In fact, Canalys puts Symbian's market share at nearer 80% based on data for Q3 2006 (see ***www.canalys.com/pr/2006/r2006102.htm***).

[4] Windows Mobile 5.0 is based on WinCE 5.1.

[5] To split hairs a little, SavaJe is not in fact 'all Java': the kernel and low-level system is written in C and the system layers are a mix of C/C++ and Java.

v9 releases will increasingly be aimed at scaling not just for the high end, where it is a proven platform for the latest feature-laden phones, but for the mid-range, mass-volume consumer market.

2.2 In the Beginning

In the summer of 1994, Psion was a company of perhaps 40 software engineers and as many hardware engineers, with a product line of handheld organizers that was highly profitable. The most recent was the Psion Series 3a, the second in the Series 3 family, a pocket-sized phone with a clamshell design sporting a letterbox format grayscale display hinged over a QWERTY keyboard, with an x86-family processor inside, up to 2 MB of RAM, removable flash memory cards and a ROM-based 16-bit operating system (named SIBO) for an all solid-state design. Its hardware design was not revolutionary but it was striking. Even more so was its built-in set of easy-to-use productivity applications. Supported by a dedicated, BASIC-like programming language called OPL, a thriving hobbyist community had established itself, self-organized (in pre-World Wide Web style; the first release of the Netscape browser appeared that same year) around bulletin boards and news groups and writing add-on software.

OPL was in fact a carry-over from Psion's original Organiser product line, which was also doing nicely, having been enthusiastically adopted as a stock control tool by UK high-street retailers such as Marks & Spencer.

That particular summer, the big project was a true Visual Basic clone (called OVAL) for the Series 3a, intended not just to increase the capabilities of the machine, but to open a bridge to the programming mainstream and tap the rich potential of the hobbyist programming market in BASIC for DOS and the Macintosh.

At the same time a much smaller project was also kicked off to create a next-generation operating system for the 32-bit devices which the company was already planning as replacements for the 16-bit Series 3 range as part of its strategy for retaining its lead in the handheld market. (In 1994, Palm had yet to release the Pilot; indeed it was still a software house, writing connectivity software for Psion's Series 3, among other things. Apple's Newton was a year old and genuinely innovative but had failed to find much of a market. Microsoft had not yet released Windows CE and the Hewlett Packard machines which were the nearest competitors to the Series 3 were based on MS-DOS and primitive in comparison.[6])

[6] In 1991, Hewlett Packard introduced the HP-95LX palmtop running MS-DOS and applications such as Lotus 1-2-3, with a 16x40 text display. It was improved to an 80x25 display on the HP-100LX in 1993 and upgraded again with the HP-200LX in 1994. Devices based on Windows CE, starting with the HP-300LX, did not appear until 1997.

The follow-on to the Series 3 was codenamed Protea,[7] and over the next year the project continued to grow. By the end of 1995 it was driving a rapid expansion of the company and in particular the project to create the new operating system (which was eventually named EPOC) was consuming the lion's share of the company's software development budget, although the Series 3 software remained in active development. For example, email and Internet extensions, in particular, were being prepared as it became increasingly clear that accessing Internet services from handheld devices was likely to become a significant market driver.

The Protea story has been told before [Tasker 2000, p. 14]. The brief was simple enough – create the next-generation successor to the Series 3, a more sophisticated 32-bit handheld to be called the Psion Series 5.[8] In this sense, then, the project was quite narrowly focused on creating the next successful product. But from the software perspective, the longer term vision for EPOC was explicit. The design brief called for it to support not just the explicit requirements for Protea applications, but the as-yet unidentified requirements for other future products. While there was as yet no talk of licensing the operating system, there was a long-term vision. The next generation, like the current generation, would be a family of products and there was an explicit intention that the software should aim for a design life of perhaps ten to fifteen years.

The Protea project delivered in the early summer of 1997. Like many complex software projects, it was late but not excessively so. However, somewhere along the way an interesting shift had occurred. By the time the all-new Psion Series 5 shipped in June 1997, the software side of the company had been spun out (as Psion Software, in late 1996) and the first licensee software projects had started.

The Series 5 was an outstanding industrial design, with a true tactile keyboard (on which you really could touch type) and a backlit touch-screen with an ingenious hinge that ensured the device remained stable when used with a pen in touchscreen mode.[9] (Competing, non-Psion products tended to fall over backwards when the screen was pressed.) A CF card slot was provided for expandability and, best of all, the Series 5 seemed to run forever on two AA batteries. As for the software, it rapidly acquired a reputation for extreme usability and legendary robustness (after some natural early teething troubles).

The Series 5 was a best-seller though, quite probably, it did not sell as well as its predecessor, the Series 3. (In its lifetime of five years of

---

[7] Protea is the name of a flower native to South Africa. As it happens, Psion founder, David Potter, and the first two CEOs of Symbian, Colly Myers and David Levin, all share a connection with Southern Africa.

[8] Actually, as David Wood recalls, for a long time it was assumed it would be called the Series 4.

[9] Credit for the Series 5's famous hinging clamshell case goes to Martin Riddiford of the Therefore design consultancy.



Figure 2.1 The Psion Series 5 MX

production, the Series 3 is thought to have sold more than 1.5 million units.[10] The Series 5 and its immediate successors including the Revo, had a lifetime of four years of production, during which it sold probably around a million units (see Figure 2.1).

The EPOC team had started with a clean slate, but the operating system did not come out of nowhere. Many of the ideas had been tried, tuned and proven in one or more, sometimes all, of the previous systems. Clean and 'from the ground up' it may have been but it was nonetheless a from-the-ground-up rewrite of the 16-bit operating system for the Series 3, which in turn was a from-the-ground-up rewrite of the second-generation 8-bit operating system for the Organiser II. (The first-generation 8-bit system for the Organiser I had only rudimentary operating system features and was, in effect, written straight to the metal.)

While, by any measure, the new operating system was written remarkably quickly,[11] the fact remains that operating systems gestate slowly and cost years of effort to create.[12] Counting from the first Organiser systems, Psion had already invested a dozen years in operating system development when the Protea project began. Planning for a design life of at least as many years for the new operating system was a matter of basic commercial common sense.

It is likely that, had Psion had been a pure software company (or just a larger and more mature company), a from-the-ground-up rewrite,

[10] See ***http://3lib.ukonline.co.uk/historyofpsion.htm***.

[11] Martin Tasker puts its development time at 3.5 years and its cost at £6 million [Tasker 2000, p. 15].

[12] There have been some interesting attempts to quantify the development cost of Linux (see for example the article by David Wheeler at ***www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html***).

let alone one using a new and unfamiliar, object-oriented language, would not even have been considered, let alone allowed to complete. The business logic would almost certainly have favored extending the existing system and Psion very likely would have missed its moment. But Psion was not a software company, nor did it really think of itself as a computer hardware company; it was a product company, driven by a whole-product vision. And what's more, it had enough cash in the bank to do what it liked in pursuit of that vision. Which is just what it did.

2.3 The Prehistory of Psion

Psion started life distributing computer hardware but moved quickly into software, capitalizing on the pre-PC microcomputer boom, writing games and then office applications for machines such as the Sinclair ZX81 and Spectrum. It was a small-scale operation in a mews behind Gloucester Place in Marylebone, London. Early hits included a flight simulator for the ZX81 and a spreadsheet application for the Spectrum. The flight simulator was written by Charles Davies, an early director of Psion, later the Chief Technical Officer of Psion and now of Symbian. The spreadsheet application was written by Colly Myers, another long time Psion director, later CEO of Psion Software and Symbian's first CEO. The legend has it that Myers wrote the complete application from scratch in the course of a single flight from Johannesburg to London.

Few people still in Symbian can trace their roots back quite so far, but someone who can is Howard Price, now a senior system architect at Symbian, who joined Psion in 1983 with a math degree, having settled in London after traveling, largely to avoid returning to military service in (pre-democratic) South Africa.

Howard Price:

In 1983 we were only doing Spectrum work, mainly games. We all sat in a row along a workbench, about eight of us, with Charles Davies at a desk near the stairs and David Potter in a little office at the end. And everybody programming in assembler, pretty much, Z80 assembler for the Spectrum, which we wrote on some HP-type machine. Downstairs were all the boxes containing our programs on cassette that had come back from Ablex, the mastering company. Once a week, a truck would arrive and everybody would line up to throw the boxes onto the truck.

Charles Davies had joined the company at the invitation of its founder, David Potter, after completing a PhD in computational plasma physics. Potter had been his thesis supervisor.

Charles Davies:

I was programming 3D models in Fortran and then I left and joined him. There were Fortran programmers who didn't know the length of a word on the CDC machines we were using, but I always had an interest in programming, so I learned assembler. And then I went to microprocessors and I programmed a lot, first Pascal and then C. I learned C on the job at Psion.

Psion bet heavily on the success of Sinclair's follow-on to the Spectrum, the Sinclair QL, developing an office suite application. It was badly jolted when the machine flopped and the software didn't sell. The surprise success which emerged at around this time was not software at all but hardware: the original Psion Organiser, the pet project of the company's single hardware engineer.

The Organiser launched Psion as a product company and manifested what became its signature traits: carefully designed hardware products whose very modest means were maximized by great software. The games had made money but the devices rapidly became the soul of the company.

Charles Davies:

We had a product vision for the Organiser. It was an 8-bit device and the software was written straight above the bare metal, so there was no operating system. The first ROM was 4 KB, subsequently 16 KB and we had a programming language in there called Forth, even in the 4 KB, because Forth is that tight that you can do that.

The success of the Organiser put enough money in the bank to fund development of a second-generation device.

Charles Davies:

The Organiser II had the luxury of a 16 KB ROM. The first product didn't have any serial port and people wanted to add barcode readers and things, so we added a serial port. But then you had to write add-on software to talk to the serial port and Forth wasn't up to it. So OPL was invented at that time, a BASIC-like language. And because of that we ended up having to document certain library routines for extending the software in the ROM. That introduced us to the idea that when we go to the next generation we need an operating system proper. We need to separate the applications from the system, because we had library routines, but no operating system separation.

A few months after Howard Price joined, the original HP machine on which Z80 software was developed was replaced by a VAX, at the time

a huge investment for what was still a very small company. When the VAX arrived, development moved from assembler to C. It was a big and exciting new language for the Psion programmers. Programs were written and cross-compiled on the VAX for the Z80 chip. But C had not been the first choice of programming language when the VAX arrived.

Charles Davies:

When we first got the VAX and decided to go from assembler to a high-level language we chose Pascal, which was the system language for the Apple Macintosh at that time. That choice lasted a few months. But then we read Kernighan and Ritchie and it was just obvious that this is a whole load better than Pascal for what we wanted to do. We recognized that C was the right language for us, because with C, you know, 'how low can you go?'. We recognized that in C, we could do the sorts of things that using Pascal we had to do in assembler. But that switch wasted money. At that time, a VAX cost £100 000 and you work out what that is in real terms now, it was a big investment and the compilers cost thousands. So we bought Pascal compilers and wasted a whole load of money and had to re-buy C cross-compilers.

Despite its expense, the VAX turned out to be a fortuitous choice. The VAX ran DEC's VMS operating system. That the early influence of VMS should eventually show up in an operating system which has become best known as a mobile phone operating system is startling at first sight. VAX, after all, was the dominant mini-computer of the late 1970s and pre-PC 1980s and VMS is in many ways a dinosaur of the big-metal era. But Symbian OS traces a very specific legacy to VMS, which indeed goes all the way back to the first Psion operating systems for the Organiser products.

2.4 The Beginnings of Symbian OS

When Geert Bollen joined Psion in May 1995, the 32-bit operating system project (EPOC) was well under way. Bollen had been in the UK for just six months, after moving on from a Belgian startup which had folded. A Macintosh-only shop with strong university links and specializing in document management systems, Bollen found Psion instantly quite different, 'a little bit homegrown', as he puts it.

The EPOC project was dominated by a few key personalities and largely divided up between the teams they had gathered around them. Colly Myers was responsible for the kernel and base layers, Charles Davies for the middleware and David Wood for the user interface. All were directors of Psion, and later Psion Software and Symbian, as was Bill Batchelor who had taken over the running of the overall Protea project.

Geert Bollen:

Most of the architecture came ultimately from the interaction between Charles Davies and Colly Myers and the creative tension between them. And sometimes it could take a long time to settle something. So they would have an argument and then out of that they would come up with something sufficiently rich for Charles, sufficiently doable for Colly and then an implementation would appear.

And once the implementation had appeared, that was very much that. As Bollen puts it, it could be extremely difficult to influence the implementation after the fact. Myers and Davies had styles which were as different as were their personalities.

Geert Bollen:

Charles Davies the cerebral purist, Colly Myers the bull-dog-like pragmatist, 'We are building the system and I have to know what to build!' So to give you an idea, Colly was off implementing the system, meanwhile Charles was masterminding a complete Rational Rose model for it.

Martin Tasker was another early recruit, joining a few weeks after Bollen when the software team was around 30 strong. Tasker had been at IBM, working on System 370 mainframes, having studied computing at Cambridge.

Martin Tasker:

I joined on 19 June 1995, 180 years and 1 day after the battle of Waterloo and 6 weeks before my wedding! There was fantastic intellect and purity of design. I think the atmosphere was really quite frontier. You got a senior position, I say 'senior' but this was a very small team, but you got authority within that team by basically stating your opinions and being seen to have good ones.

There was Colly Myers, in those days with a beard, who would sit there talking and he just couldn't keep still, he would be twitching.

And there was Charles Davies, face raised at a Victorian angle.

Commitment was total and the dynamic could be abrasive. Strongly held opinions, strongly defended, were part of the culture.

Martin Tasker:

Colly Myers was a combination of charismatic leadership and utter frustration!

Tasker indeed recalls an incident from the early days of the project, a small but significant difference of views between Myers and Davies which turned, as such differences usually did, on weighing the balance between purity and pragmatism.

Martin Tasker:

Colly Myers had a theory that array arguments should be unsigned, which meant that a descriptor length should be an unsigned value, in other words a `TUint` argument. And I well remember a meeting in the first floor corner office of Sentinel House, which was the operations room for the project and Charles Davies' exact words were, 'I'm having a bit of trouble with the troops'. They were unhappy about using unsigneds, because there are all kinds of things you might legitimately do when manipulating descriptors which would result in a signed value as an index, for instance you multiply by some signed value. So that plays havoc with the array. because you're getting all `fffff`s as an index.

And what this triggered in Colly! 'Are you mad?!' You know, 'Who are these programmers of yours?' So the abstraction is that an array has only a zero or a positive index, it cannot have a negative index. Therefore, what possible advantage could there be in allowing a signed index? So Colly's line was, 'You just need to teach your programmers how to program!'

Well I remember four weeks of totally polarized debate and heated argument. Of course Colly was right, but the fact is that it's just too hard to do the right thing. So I walked in one Monday morning and checked out a release note, this would have been October 1995, it just read 'As agreed, changed all `TUint` to `TInt`.' Of course we kept `TUints` for flags and such – but as for numbers, that was how that debate got closed. And Colly's 'agreement', when it came, was characteristically unilateral, announced through the release note after a gruelling weekend's work which couldn't really be automated – Colly really had to check every change manually.

The relentless development pace and constant project pressure were hard, too. Peter Jackson, who these days is responsible for Symbian's software configuration management systems, remembers the approach to project management, as directed by Bill Batchelor, with mixed feelings.

Peter Jackson:

Bill Batchelor liked getting his hands dirty, but then he became project manager for Protea and so he had a dual nature. He was passionate about the right things, but at the same time he didn't like it when you told him how long it would take to do 'the right thing'.

But the company was vibrant and small and people didn't actually mind hacking away for all hours of the day and night to achieve the end goal. Bill's over-optimistic project plans were just part of that mix. He would cajole

you into committing to something that was impossible, and you'd do your best to achieve it, and then eventually there would be this undercurrent where everybody knew it was totally absurd, but no one was going to say it. Eventually it would all come out. And then there would be another planning iteration and the same thing would happen again.

The big practical problem with the Protea project, one which caused a succession of headaches for Geert Bollen, was the lack of real hardware. Software development started well ahead of the availability of any prototype hardware but even by mid-1995, when the software project was in full swing, the device prototypes were still not ready.

Geert Bollen:

The on-the-metal version of the kernel was started and delivered after I arrived and Colly Myers assembled a team for that. Before that Colly had been a one-man band. The GNU tools at that time were coming on. I had some involvement in that but they were still a long way from being rolled out.

Andrew Thoelke is another veteran who joined in March 1994 and is now the Chief Technology Architect at Symbian for the base services and kernel layers of the system. In the absence of hardware prototypes, built code was run on PCs using an emulator layer which mimicked a full system by mapping low-level operating system calls to their Windows equivalents, essentially the same approach used by Symbian OS developers today in the first stages of development, before moving to hardware targets.

Andrew Thoelke:

Down in the base team, not having hardware was a problem, so the system was first brought up on x86 as a hardware port before it was ever brought up on ARM. In the original kernel architecture, probably 40% or 50% of code is shared with the target, but there's still vast amounts of kernel code which is target only, all of the scheduling and threading, the interrupt model, the device-driver model, so all of that needed to be done with a real target. So they used a 486, they basically built an 8386 port of the system first, because that brought online another 40% of the kernel code. Obviously there was still ARM-specific hardware code and a different MMU and all that sort of thing. But it was actually much less work when hardware did become available because they had already got a generic kernel mostly working.

Whatever the problems, there was no doubt in anyone's mind that what they were creating was special. Martin Budden, now Chief System

Architect at Symbian, was a veteran of the two 16-bit projects before moving onto the EPOC project. He puts it very simply.

> **Martin Budden:**
>
> I came to the company because I wanted to do something that was exciting. As soon as I saw what Psion was doing, I just knew that was what I wanted to do.

Looking back, Psion's timing was good; it had judged the moment perfectly.

> **Martin Tasker:**
>
> Psion, like many companies then and not just in Britain but elsewhere, had achieved success by innovating according to rules which nobody had ever written. It just did its own thing and it found a niche in the market.

2.5 The Mobile Opportunity

When the Psion board decided to spin off its software division, which at that time numbered around 70 engineers, it was effectively a public commitment to a software-licensing strategy.

It is clear that a number of different options were considered. There were rumors at about that time that Psion had considered buying Palm. A possible purchase of Amstrad got as far as due diligence. The background is revealing. For the Psion board, the real target seems to have been a Danish phone-making company, Dancall, which Amstrad had previously bought and absorbed into its empire. Thus, buying Amstrad would have enabled Psion to become a phone manufacturer. This indicates very clearly the direction in which Psion was pressing at that time. In the event, Psion did not buy Amstrad and Dancall was eventually sold to Bosch, before being sold on to Siemens. Much later, it was the formerly Dancall site at which the Siemens Symbian OS phone, the SX1, was developed. Still more recently, the site has been sold on to Motorola.

Psion, of course, did make its move into the phone market, but in a quite different direction. It was a visionary move and one for which the company founder David Potter deserves enormous credit. There were other visionaries too. In particular, Juha Christensen, Psion Software's

bravura marketing director,[13] had assiduously begun to cultivate mobile phone manufacturers, Nokia included. Psion Software was certainly not their only choice of partner for collaboration at the time (just as Symbian is by no means the only choice today). However, the company was perfectly positioned, with just the right product at just the right time in the evolution of the mobile phone market. It has succeeded remarkably well in extending that early lead into a commanding position in the market.

2.6 Background to the First Licensee Projects

The first Organiser shipped in 1984. Over more than ten years, Psion honed its hardware and software skills and learned through three complete iterations (Organiser, Organiser II and Series 3) what it took to create a complete software system for mobile, battery-powered, small-footprint, ROM-based systems, before embarking on the 32-bit EPOC operating system from scratch. The Series 5 shipped in June 1997. Almost exactly a year before, Psion's software division had been spun off into a separate company, Psion Software. Almost exactly a year afterwards, in June 1998, Symbian was created as a joint venture aimed at bringing EPOC as a new operating system to mobile phones.[14]

Even before the Series 5 project completed, licensees of Symbian OS from at least three companies were waiting in the wings. There are different versions of the story, but they all agree on the main points.

Martin Budden:

As I heard the story, Nokia were in the market for a new operating system for their Communicator and they approached us. I know that Juha was instrumental in brokering the deal, but it was Nokia's idea and I remember there was a time when we were told Nokia were coming to see us. It wasn't exactly 'smarten up the office', but you know, 'if they ask questions, give good answers'. It was Nokia that was strongly in favor of bringing in other phone manufacturers to form a consortium, or that's what I understand. They fairly quickly brought Ericsson on board and then Motorola got on board at the last minute, and that was also quite significant.

[13] The legend within the company when I joined in 1997 featured Juha going off to cold Northern climes, sharing saunas and vodka with Important People and coming home with a Nokia deal in his pocket. Juha was later tempted away to Microsoft to lead the Windows Smartphone effort.

[14] [Tasker 2000, Chapter 1] provides a definitive history.

'Symbian Day' was June 24th 1998 and the Psion share price rocketed on the news (causing much excitement in the office over the next few days). A few days before, we had delivered the first free SDK for what was still Release 2 of EPOC.[15] Version 5 was still a whole year away and the first Unicode release, ER5u as it was known, was a step further still.[16] This, arguably, was Symbian OS 'version 5' (although it was never called that), the first operating system release that was 'fit for phones', although even then the first phones were still a year away from market. The first designated Symbian OS release, v6, appeared in Spring 2001.

From a public perspective much was made of the Nokia versus Microsoft angle and some commentary viewed the creation of Symbian as an attempt by Nokia to build an alliance against Microsoft. But it seems just as meaningful (and more useful) to view it more as a case of Nokia making a shrewd move to work with competitors to consolidate and grow a new market (that of highly capable, multi-function, phone-enabled terminals: 'smartphones' in other words), at the same time enabling Nokia to focus on what it clearly saw as its strength, the user interface. The evidence [Lindholm 2003] is that Nokia viewed the user interface as the critical software design factor for the phone market – if not the key determiner of success then at least a critical one – and also that it viewed the user interface as its critical strength.

However, even before the Nokia approach, Psion had been actively evolving its own strategy and there is no doubt that a fundamental shift occurred after the start of the Protea project, leading to the spinning off of the software division to open the way for software licensing. The team led by Howard Price moved across quite late to the EPOC project to start what turned out to be the last rewrite of OPL, this time for the 32-bit platform. By then the company's focus had shifted quite noticeably.

Howard Price:

The big thing in every team meeting was, 'Where are we with licensees?' So we'd go to the senior team brief and a lot of the talk would be about winning another licensee, or that the licensees were getting unhappy because of this delay or that delay, or that they were worried we were delaying their products to concentrate on Psion work – the Series 5 project was running worryingly

[15] Giving SDKs away to developers was still considered controversial within the company at that time.

[16] ER5u was an interesting experience to live through, a complete rebuild of a system which still, at that time, did not routinely build from source (as it does these days, with nightly builds of multiple variants from a single master codeline), with the 'wide' flag set for all components in the system so that all descriptors, text data and resource strings (anything with text, in other words) built 'wide' using multibyte (UTF-8) Unicode text encoding. A complicated system of 'baton passing' was evolved to follow the dependency graph up through the system and ensure that for every component, all dependencies built first; not trivial in a system which still harbored some awkward circular dependencies.

late. It took a year longer than planned for us to ship and the licensees were waiting.

The licensee strategy was squarely pitched at the phone market.

Andrew Thoelke:

The Series 3 family had been Intel based, but even at that point in '94, '95 the view was to migrate towards mobile and cellular applications. The jump to ARM was intentional, because Intel was clearly not a player in that field and ARM was already doing well and had ambitions to become far more important in that space, so it was quite a strategic move. And part of the mindset behind the next generation operating system was to target ARM. Even at that point David Potter could see that handheld computers and PDAs and cell phones would converge. And that's why in '96, before the Series 5 was actually shipped, Psion put its software division out into a separate company, specifically so that it could look at licensing its software externally.

The very first of those early collaborations was a project to create the software for a mobile companion device for the Philips Ilium phone. The companion and phone clipped together back to back and connected through a hardware slot on the back of the phone, turning it into a PDA/Communicator with 4 MB RAM, a Series 5-sized landscape-mode touch screen, a choice of soft (on-screen) keyboard or handwriting recognition and a full PDA application suite including calendar, organizer and contacts book. Communications functions included email, web, fax, SMS and full voice calling.[17]

The software was based on the November 1997 Message Suite release of EPOC (also used in the Series 5 mx), which added email and web applications, dial-up networking and TCP/IP, the C Standard Library and the Message Suite itself. The project was publicly announced as the Philips Ilium/Accent and showed at CeBIT at the end of 1998, but it never came to market.

Martin Budden was the technical lead on the project, which involved writing not just a bespoke user interface, but also a complete applications suite including messaging and contacts applications. As he says, it was a significant amount of work. However, compared with later projects, these were very much toe-in-the-water exercises, both for Symbian and its licensees. On the Symbian side, the team was relatively small, perhaps a dozen developers working on the user interface and applications and half as many again working on the software port to new hardware.

[17] The Ilium is described at ***www.noodlebug.demon.co.uk/goingmob/spphiila.htm***.

The Philips device was the first licensee project (unless Psion itself is counted as a licensee) but, most significantly, it was the first project to generate licensing revenue in the form of pre-paid royalties.

Martin Budden:

The Philips project was the first bit of money that we ever got in from licensing; the first licensed product we ever got and made some money from.

Other licensee projects followed, including the Series 5 look-alike Osiris from Oregon Scientific and the Geofox One, a keyboard-based PDA with a larger keyboard and screen than the Series 5 and with a laptop-style touchpad instead of a touchscreen. It also added a built-in modem and a standard Type II PCMCIA slot.

While they demonstrate the enthusiasm with which Psion Software set out to develop a licensing model, all of these projects were ultimately false starts, failing to capture much attention from the market. Straight after the Philips project, Budden moved onto another small licensee project, working on behalf of Ericsson, and stayed as technical lead through the project startup. While the biggest problem in the Philips project had been trying to work around the limitations of the hardware design, which had been more or less fixed before the start of the project, the Ericsson project was a true phone design. In particular, the hardware design provided a robust solution to the problem of communications between the phone hardware and the application processor. As Budden says, the feeling on the team was that the hardware design was right from the start.

The fact that it was another phone project indicates where Psion saw the market opportunities, but it also indicates the direction in which the licensees saw the phone market moving. The goal of the Ericsson project was to create a mobile phone with full PDA functions, as full as was then possible. The result was the Ericsson R380, a breakthrough product not because it sold particularly well (it was probably too far ahead of both the market and the current state of technology) but because it rehearsed key principles which led the way to later successful Sony Ericsson Symbian phones, starting with the P800 and followed up by the highly successful P900 family of phones.

Biggest by far of all these projects was the collaboration with Nokia to create the Nokia 9210 Communicator, which started while the Philips project was still running. While the Ericsson R380 team had roughly double the numbers of the Philips project, the Nokia 9210 project eventually involved probably half the company; by the time it completed, the company had grown from 70 to over 200.

The Nokia 9210 project completed after that of the Ericsson R380, but began before it. Earlier projects and, to some extent the Ericsson



Figure 2.2 The Ericsson R380



Figure 2.3 The first Symbian phone, the Nokia 9210 Communicator

R380, had been based on snapshots of the evolving operating system (which was still named EPOC), with the deepest changes concerned with the adaptation to new hardware and bespoke customization of the user interface. In contrast, the Nokia 9210 project drove a complete iteration of the operating system from the ER5 baseline to what became known, finally, as Symbian OS v6.0. Conceptually, the Nokia 9210 was the first Symbian phone, even though it wasn't the first to market (see Figure 2.3).

The transition from the Series 5 to the Nokia 9210 was less a series of steps than a route march, four years of hard work (from inception to completion). Symbian OS has been (and will no doubt remain) a continuous evolution towards a destination which is always one twist of the road away.

2.7 Device Families

The native EPOC graphical user interface (GUI), which defined the look, feel and interaction style of the device software, was known as Eikon. Eikon was designed for extensibility and customization. However, the

extent of the variations required by different customers, driven by the needs of devices that, increasingly, were not PDAs but phones with PDA functions, significantly exceeded the assumptions of the original design. Each project effectively created a complete bespoke user interface, albeit from the common starting point of the Eikon code. Not only did this level of customization not scale, it was clearly threatening to fragment the platform.

Martin Budden:

The model of doing a bespoke user interface was already there. We did a bespoke user interface for Philips and for the Ericsson R380. And for the Nokia 9210 Communicator, again there was a new user interface to Nokia's specifications. But there were fundamental conflicts between these user interfaces, in practical terms of 'Did they have pens?' or 'Were they keyboard based?' and 'What was the screen size?', but also in deeper terms of the whole user interface philosophy and what you expose to the user. And it just became clear that if we did a user interface for every single phone, that wasn't going to be sustainable.

Symbian's solution was the so-called reference design strategy. The specific phone types were genericized to reference specifications: in practice, that meant the keyboard-based Communicator-style device and the pen-based 'smartphone' equipped with PDA functions and based loosely on the Ericsson R380. As well as a form-factor definition specifying the essential features of the physical design and therefore, in effect, parameterizing each design to a particular market point (in terms of features, size and key use cases), Symbian would supply a generic user interface for each form-factor, which licensees would then customize.

As realized in Symbian OS v6.0, devices were identified as 'smartphones' (phone form-factor devices) and 'communicators' (PDA form-factor devices). Communicators were further divided into keyboard-based (the Nokia 9210) and tablet-based devices (both Ericsson and Sanyo showed off prototypes broadly similar to Palm or Windows CE devices such as the Pilot and iPaq). Two reference user interfaces were included in v6.0 as 'device-ready' designs: Crystal, which shipped with the Nokia 9210 and which eventually became Nokia's Series 80 user interface; and Quartz, which eventually evolved into UIQ.

A number of other device family reference designs (DFRDs) were proposed and several proceeded to reasonably advanced specification, including Sapphire which was split into Red and Blue variants, depending on screen size and interaction mode (pen or keypad); Ruby, which evolved from Red Sapphire; and Emerald, which encapsulated the original smartphone concept as realized in the Ericsson R380. Neither Ruby nor Emerald were announced or came to market. Blue Sapphire eventually

evolved into the Pearl DFRD and finally reached market branded as the Nokia Series 60 user interface (see Figure 2.4). Pearl had first been defined as a 'headless' DFRD (without a user interface), before acquiring code branched from Crystal and eventually unifying with the work which had been going on independently within Nokia to develop what was known as the 'square' user interface.[18]

Pearl in effect became the first true smartphone platform (defined as a phone with information capabilities) and was realized in the first Series 60 device, the Nokia 7650.

Quartz, meanwhile, never came to market in its original form, that of a tablet-style device most closely resembling a phone-enabled, pen-orientated PDA, which was dubbed the Mediaphone reference design when prototypes were shown at CeBIT in 2001. The Quartz design had originated at Symbian's Ronneby site in Sweden. Originally an Ericsson development laboratory specializing in Windows CE devices, the site had been transferred to Symbian as part of the original Ericsson investment in the consortium. Quartz quite clearly inherited Ronneby's design legacy. However, the device format with which Quartz did eventually come to market, by this time rebranded UIQ, was the one pioneered by Ericsson with the R380: pen-operated, a screen that could switch between portrait and landscape modes and with a key-pad flip. In, first, the P800 and then the P900 (see Figure 2.5), this format has become a signature design of Sony Ericsson's high-end, business-orientated range and has been extremely successful.

The Crystal user interface of the Nokia 9210 Communicator was eventually rebranded Series 80 (see Figure 2.6) and remains the basis for the product line which continues to evolve and innovate. (A Communicator was the first Symbian phone to offer Wi-Fi connectivity, for example.)



Figure 2.4 The Series 60 user interface, as used on the Nokia 7650

[18] David Wood believes that Nokia's work on 'square' had been in progress for at least two years before the formation of Symbian.

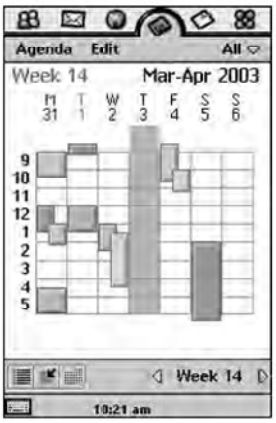


Figure 2.5 The UIQ user interface, as used on the Sony Ericsson P900



Figure 2.6 The Series 80 user interface, as used on the Nokia 9500

While a number of licensees worked on Quartz devices and others (besides Nokia) expressed interest in Crystal-based devices, the DFRD strategy eventually stalled. The reality is that, although they aimed to be generic, the designs could not escape the pull of licensees. In the end, they were more licensee-specific than generic, reflecting particular licensee's views about what a phone should be. Nokia drove Crystal; Ericsson and then Sony Ericsson drove Quartz; and Sapphire seemed to split and split again, until there was a one-to-one mapping between DFRDs and licensees.

Martin Budden's view then and now is that the problem was essentially not resolvable. It was not possible to agree on a Symbian-based user interface, in other words one evolved from the original Eikon GUI of EPOC and the Series 5, which was suitable for the different device visions of Nokia, Ericsson and Motorola.

And this was the problem. Each phone vendor had different, deeply held views about what makes a phone. Symbian was trying to create a

software platform that would satisfy them all. Motorola wanted a pen-based user interface. Nokia wanted a keyboard-based user interface. Nokia did not place much value in the power of the pen and to date have only ever released one pen-based range of Symbian OS phones, the Series 90 user interface on the 7700 and 7710. Quartz, coming from a design unit which had started out working on Microsoft Windows CE devices, came up with a pen-based tablet format, such as the Compaq iPaq or the classic Palm devices.

Not everyone in the company was convinced by the DFRD strategy. To some it seemed more like an attempt to paint over the deeper problem, that conflicting licensee user interface requirements were irreconcilable.

Martin Budden:

In my view, we just could not resolve the issue. We couldn't come to an agreement on what the Symbian-based user interface would be that was suitable for all licensees, which I think was ultimately why Nokia went off to do their own.

The designs were not so much generic as licensee-specific, reflecting each licensee's views about what a phone should be.

Martin Budden:

The DFRD idea was to have families of user interfaces, so there would be one family for devices like the Nokia 9210 Communicator and the Series 5; there would be a family that was based on the Ericsson R380 which was a phone; and at about this time, Quartz started up and that was for another Ericsson phone with quarter-VGA tablet form-factor. There was a basic conflict there between Nokia and the Series 5 user interfaces, because what Nokia wanted went further than the customizations the user interface could easily accommodate and then the other conflict that started to manifest itself was a user interface for a phone form-factor smaller than even the Ericsson R380.

After his time on the Ericsson R380 project, Budden had moved across to work on Quartz as technical lead and spent the best part of a year commuting between London and Ronneby. He witnessed the difficulties of implementing Quartz as a product-ready, concrete instance of a DFRD at first hand. Partly the problem was one of resourcing, with the company's main focus dedicated to the underlying features of the operating system, many of which were driven by the ambitious requirements of the Nokia 9210 project. The fact that Quartz was being

developed at a remote site did not help. Nor was it necessarily easy for the Ronneby engineering team to adapt itself to the very different style of Symbian, of which it was newly a part, compared with its previous parent, Ericsson. For its part, Symbian probably found integration of a new remote site just as painful. Beneath it all, were the basic engineering problems.

Martin Budden:

The Quartz team had very great difficulty getting anything done which would support their work and that led to a lot of fragmentation and reimplementation and it also highlighted that there was a lot of code that was not easily separable. So, for example, the messaging code had user interfaces in the engine layer, which meant that to change the user interface we had to redo a lot of the engine code as well. So there were a lot of things that made it difficult to separate out the bits. And this is when the idea of resolving all these conflicts by defining DFRDs emerged. I thought at the time that it was never going to fly.

However, the strategy served its purpose as, between 2000 and 2002, it enabled the important focus to become that of developing the underlying operating system. It also underwrote the splitting of generic from specific functionality in Eikon, the original EPOC GUI, as a necessary engineering step to enable the creation of the DFRD variant implementations.

When it abandoned the DFRD strategy in 2002, Symbian made a tactical retreat out of the user interface space altogether. The Pearl DFRD which was being actively developed in collaboration with Nokia, was taken over entirely by Nokia to become Series 60. Quartz, which by this time was the basis for projects with both Sony Ericsson and Motorola, was spun out into a new Symbian subsidiary, UIQ Technology AB, based at the Ronneby development site. UIQ became the name for the user interface.[19] Japanese licensees working under the FOMA umbrella, as DoCoMo's new 3G network was branded, went their own way and collaborated on a common user interface known as MOAP.[20]

Symbian's strategy since 2002 has been based on the concept of a 'headless' delivery to its customers with a custom user interface integrated as part of the product creation lifecycle, either by the phone vendor (in the case of the FOMA licensees and Nokia) or by a user interface vendor

[19] In late 2006, Sony Ericsson announced that agreement had been reached for it to acquire UIQ Technology AB from Symbian.

[20] FOMA is DoCoMo's 3G network, which launched in 2001. Mobile Application Platform (MOAP) was originally an internal designation which has now begun to appear in public forums.

(UIQ licenses its own user interface pre-integrated with Symbian OS to customers such as Sony Ericsson and Motorola; similarly, Nokia's independent Series 60 business unit licenses S60 preintegrated with Symbian OS to customers such as Samsung and LG).

Symbian OS is GUI-centric in the sense that the user model is exposed only through the GUI, while being designed into the operating system at a deep level; nonetheless, Symbian does not provide its own shippable GUI. This model has its challenges (as any model does) but, with the sales record as it stands, it can be considered proven. It is driven by the recognition that the mobile phone market is quite different from the desktop computing market, for example, in which users (as well as vendors) seem reconciled to the greater than 90% domination of a single user interface, Microsoft Windows. There are alternatives, in the form of Macintosh and Unix/Linux, but these are not mass-market alternatives. The attempts to create mass-market alternatives (BeOS, for example, in the late 1990s) have been spectacularly unsuccessful.

Personal mobile devices, from phones to music players to cameras, are very different. Consumer devices, from TVs and DVD players to Hi-Fi and radio to car dashboard controls, are very different too.[21] What users want from them and how they wish to interact with them, is quite different from what they want from the beige box underneath the desk.

2.8 Operating System Influences

While the user interface defines a system from the perspective of end-users and translates the design philosophy of the system into tangible behavior accessible to users, the real character of an operating system is defined at a deeper level, by the fundamental choices its designers make – typically in the form of trade-offs between aspirations and limitations, whether of performance against price, features against time-to-market or innovation against familiarity. As a company of engineers rather than computer scientists, it seems that Psion absorbed multiple influences, but then proceeded from immediate practicalities, almost as if no theory existed.

Its first true operating system was written for the 8-bit Organiser and then re-written for the next generation of 16-bit Organisers. Since these were x86-based, DOS was a possible alternative candidate. The company however decided to follow its own path.

[21] Christian Lindholm provides a fascinating public insight into the issues driving user interface design for phones as we have known them, and for the multi-function mobile terminals they are becoming. These devices, as he says, are evolving from impersonal objects to intimate possessions 'containing one's most important data and thoughts' [Lindholm 2003, p. 154].

Charles Davies:

We considered using DOS and rejected it, which was controversial at the time because there was a view that DOS was the only kind of operating system for PCs downwards and of course it was also what the HP200 used, which was our big competitor at that time.

But DOS is a strictly synchronous system, single-user and single-tasking and it also provides an interface which insists on dragging the user down to the machine level, which was not the vision at all. In fact Psion did later release a DOS-based version (MC600) of the MC400 laptop, precursor to the Series 3, just as it had released a DOS-based version of the HC handheld. As David Wood remembers, after that first-hand experience of MS-DOS, it became known in the company as 'MS-dogs'.

In contrast, the exposure to VMS was critical because it showed that there was another way.

Charles Davies:

From that experience of cross-compiling on VAX for the ZX80 we got to know VMS pretty well. It was a multitasking operating system with asynchronous services. Colly Myers was the architect and we decided to make our system pre-emptive multitasking.

While these ideas began to influence the Organiser operating systems, they became central to the design of the next iteration 16-bit system. This was the operating system which eventually made its way into the Series 3, from its first iteration for the MC400 laptop.

Charles Davies:

The strong drivers were firstly, 'always on', in other words, no bootup – the idea that the operating system ran forever – and, secondly, that you could switch from one application to another without waiting, which was the multitasking. One of the early design principles was that we started to play with servers and that was a direct VMS influence. So if you have a multitasking operating system and if applications are executables in that operating system, then the number of applications is extensible and you can switch between those applications without exiting the current application. We take multitasking for granted now, but DOS PCs at that time were single tasking, you ran an application and you had to exit it to run another application. Remember TSRs, Terminate And Stay Resident programs which allowed you to switch in and out? We thought that was crazy! We thought the main system should be like that.

The VMS influence was sufficiently visible in the final system to be noticeable, for those who cared to look. Peter Jackson had worked with VMS while at BP Exploration and recognized the influence even in the first iteration 8-bit Organiser system. He quickly became a Psion fan.

Peter Jackson:

I was attracted to Psion in the first place because I got hold of the internal documentation for the Organiser. This was the 8-bit operating system and I read the documentation and thought, 'That looks familiar. This has been influenced by VMS'. And I also thought, 'This is very clever. This is a company that could be worth paying attention to'.

The Organiser operating system displayed many of the properties of consistent and elegant design that Jackson had admired in VMS and had found wanting in other systems, for example Unix.

Peter Jackson:

I would characterize Unix as being something that wasn't really designed, it evolved. So, by contrast, VMS was a system that was much more carefully designed from the beginning and it was carefully designed in parallel with the emergence of the VAX hardware architecture. So when you look at the design of VMS, it's quite rigorous in terms of API definition and quite consistent in terms of patterns of use of those APIs. For example, typically in VMS all APIs are asynchronous and they all have ways of monitoring the outcome of a request made to the operating system or to the I/O system and so on. So once you've learnt a corner of the VMS system and you want to explore another corner of it, you don't have to climb the same learning curve all over again. And that level of consistency applies all the way up to the command interface.

By the time Psion produced its first 16-bit systems, the key VMS-inspired patterns were well entrenched.

Peter Jackson:

It was harder to get into the 16-bit system, but there was still that consistency and elegance in how they had implemented things and I would say that the attention being paid to asynchronous interfaces was a good example of that. The whole event-driven programming model was very strong in Psion in those days.

Jackson attributes those patterns not just to VMS, but also to a more general mainframe influence. While the hobbyist culture typified by

CPM and early DOS gravitated upwards from micros to PCs, the more sophisticated professional programming culture of multiple-peripheral, multiuser, 'big-iron' computing began to drift down towards smaller systems, first to minis (including VAX, as well as the PDP family on which Unix first evolved), micros and then PCs, but also to new classes of device such as those being pioneered by Psion.

Peter Jackson:

On mainframes and mini-computers, such as VAXs, where everything is event driven, you don't put in a synchronous I/O request. You don't say 'write this data to disk' and have the call return when the write is completed. You issue an instruction to write data to the disk and you get on with the rest of your life and at some point you'll get notified that the write is actually complete.

The asynchronous, server-based model has evolved to be one of the primary patterns in Symbian OS. Servers provide for serialized access to shared resources and are used throughout the system, wherever multiple users (client programs, including other system services) require access to a system resource, whether a logical resource such as a process or a physical resource such as the physical device screen (or screens).

For Jackson, the visible influence of VMS on the Psion operating system seemed to present a perfect opportunity.

Peter Jackson:

People who learned how to program on PCs, specifically on DOS, came from a different culture where that asynchronous, event-driven model did not hold. There were people that knew about event-driven programming and there were people that didn't. So I had this vision that said, all my mainframe expertise now applies here, I understand stuff that these PC programmers aren't at all familiar with. And the way I saw it then, when I was coming into the company, is that there's a whole set of software idioms to do with mainframe computing, and the way technology was evolving meant that you could now apply those idioms to smaller and smaller devices. And I was able to capitalize on my knowledge of mainframe software architecture.

The 16-bit system was a classic C API operating system, exposing a small number of system calls as C functions. Higher up in the system, object-oriented ideas had been widely applied. Initially, the operating system appeared in the new laptop-like product. Jackson recalls it with the enthusiasm of a convert. The MC400 in his view was way ahead of its time in terms of its hardware design, a perfect match for an operating system which was also distinctive and innovative.

Peter Jackson:

It was a really nice machine. It looked like a laptop computer, only slightly thicker than laptops are today, but on almost every face of this thing, every way you turned it you'd find an interface that you could plug something into. It was way ahead of its time. There was a speech synthesizer sound module that was quite sophisticated; it had a module that was a built-in modem; it had a superb keyboard and a long battery life. You could put batteries in it and it would be good for 90 hours and this was in the era when for laptops, three hours was your limit.

Unfortunately, the MC400 didn't sell, which was a substantial set-back for the company. However, Psion responded with a complete software overhaul (to reduce ROM size and improve all-round performance) and with a new hardware product, the Series 3. Significantly, the Series 3 played on the company's core competence in creating and marketing compelling small systems, which also avoided the DOS (and soon-to-be Windows) mainstream.

In software terms, the Series 3 gave Psion a second chance to prove the merits of its new 16-bit operating system. The most obvious conclusion to draw from its subsequent success seems to be that, while the market was prepared to embrace a novel design in a new device space, it rejected innovation when it conflicted with the incumbent standard, which, in the case of the laptop-like MC400, was DOS.

David Wood also believes that the MC failed because it simply had too many flaws, both hardware and software. It was less a question of standards than of quality and fitness for purpose. The digitizer was awkward to use, for example and the machine would sometimes reset as a result of electrostatic discharge when the user touched the removable media door.

Whatever the reason, its failure was compensated for by the huge success of the Series 3.

Peter Jackson:

The Series 3 used the same basic software technology and it was fantastically successful. Without the Series 3, none of us would be where we are today. So it was all to do with packaging, what device the software was packaged in.

The architecture of the early Psion operating systems and the patterns which have evolved from them and in their turn influence Symbian OS, can be traced back to a few key principles.

Charles Davies:

If you start with the idea of going from one application to another and having processes and tasks, processes more than tasks in fact, then you say, well those processes are running concurrently, they're running all the time, even though only one of them may be on the screen. And by the way we did produce devices like the MC400 where multiple applications were visible on the screen at the same time, though we ended up going toward smaller devices. But the vision was that the user could see multiple applications at the same time. We recognized that all of those applications would need to access a file system or whatever, that if you were running multiple applications at the same time and they compete for the same resources then you need to sort that out. One way of sorting that out, which is the design pattern we adopted, was to use a server to serialize access to a shared resource. So that was the slogan, 'servers serialize access to a shared resource'.

And the mass-storage media card on the devices at that time was a shared resource too, so you couldn't just have applications writing to mass storage, you had to have something in between that was sorting out that access. VMS used servers for that, a file server, so we made a file server and we also had a supervisory process server. So the design principle of the 16-bit system was that you had client–server and fast context switching, so there's no penalty for using servers and you get a clean architectural way of serializing access to shared resources, including memory, which I suppose is what you could say the system supervisor was.

The server principle runs deep and important design consequences follow. For example, the need for fast context switching is what determines the process and thread architecture of the operating system.

Charles Davies:

So to grow the heap dynamically, well you had a server process to do that, which is the modern pre-emptive multitasking kernel approach. And a file server and the idea of servers for other things. We knew we wanted to do graphics and we wanted to have a windowing system. The competition was still doing character-mode graphics at that time and we wanted a true graphics mode, with variable fonts and more than one application drawing to the screen at the same time. And so we had to have a model for doing that. So we said, 'Okay, the file server shares access to mass storage and the window server is the right architecture for serializing shared access to the screen.' And you need a windowing system. Even on phones, other processes can pop up a notification at any time and it basically handles that. So that was an advanced attribute of the design.

The design principles were not necessarily novel but their application to the class of small device that Psion was pioneering certainly was, the

Apple Newton notwithstanding. The vision that Psion was pursuing so hungrily was of sophisticated pocket computers aimed at an audience of consumer users rather than technical wizards – mobile and pocket computing for all.

Charles Davies:

There were GUIs around at that time, Amiga and Macintosh, that did clipping. Windows was tiled at that time, but we said we wanted overlapping windows. We couldn't afford a hardware solution. We had been involved in doing an abortive piece of work for Thompson, which we called a Thompsonitosh, which was Macintosh-like with hardware support for overlapping windows, so we knew about those things. We had also worked on software for the Sinclair QL and we'd worked on PC software. Remember, this was the age of integrated suites, which at that time were still character-mode-based and came with their own windowing APIs. IBM, for example, had something called TopView at the time. So that was the kind of environment, but we wanted to do graphics and we wanted a contemporary, modern way of doing it.

So we had pre-emptive multitasking for windows and the Window Server was born and lots of things got done in servers. The idea of fast lightweight client–server internals and servers managing clients were early design principles. The other part of client–server architecture, I guess, was the idea that this is an operating system that needed to run for years at a time without a reboot and that meant you had to have system software that looked after badly behaving applications and so that led to the idea that servers managed their clients if the clients didn't do the right thing, so that the servers didn't get left with memory leakage or data from long-gone clients. Servers knew about client processes and managed their data on behalf of client processes, even if they terminated, so there are services to let you know if a client dies and also to clean up server-side resources so that you could run for a long time, because for sure you were going to have many dying applications panicking over time.

Another design principle that came in the early days was asynchronicity. We learned that from VMS, the idea that you had event signaling and not polling. So part of that was that we were designing for battery-powered devices and ROM-based devices, which is why we thought DOS was not appropriate because it wasn't designed for either ROM-based or battery-powered systems. For us, 'execute in place' was the norm – the idea that you executed in place in ROM but you could add applications that loaded – that was part of the design. The idea of dynamic libraries was an early part of the design and I guess we were aware that Windows had dynamic libraries. We knew we had to have shared libraries and we didn't want to be loading multiple copies of the same code, which by the way was the norm if you just used compilers and linkers in the usual way. I mean these were times when people had overlays, where the code got loaded in at the same addresses, right? That was when Bill Gates famously said 640 KB should be enough for everybody. And having written software with the overlay model, we figured we didn't want that. Overlays were impossible to manage. They fell over under their own complexity after a time. So we wanted libraries that were loaded once.

These principles were rehearsed through the three generations of Psion operating systems leading up to the creation of EPOC and eventually of Symbian OS. But they were driven also by a product vision. The company was driven by the vision of creating products aimed squarely at ordinary users, which would entice them and charm them and become indispensable pocket companions.

Charles Davies:

We were building products. We were working from an idea of the user experience that we wanted. So we didn't just do pre-emptive multitasking because we thought we wanted to do an operating system and that was the fun thing to do, although there was that element to it too, if we're being honest. But we had a vision that you shouldn't have to wait for boot up, that this would be an instantly available, instant-on device and one where you didn't have to exit one application before you could run another one, because that wasn't an appropriate user experience for a handheld device.

We also thought that multitasking was a good thing for writing robust software. We had this ethic of robustness, that the product didn't go wrong and that you didn't have to be a techie to use it. Because in those days you know, I remember the first 5 MB hard disk we bought for £6000. £6000! And you went on a training course to learn how to use it! And that was not the vision of the product that we had. We had a vision of a product used by somebody who wasn't stupid, but who wasn't going to read the manual, a device where the operating system did the work for you rather than the other way around.

So it was based from the user experience backwards; the technology was in support of the user experience. That was in the bones of the product vision. We didn't think of ourselves as producing an operating system and an application suite. We thought of ourselves as producing a product that would sell. It would walk off the shelves because people wanted it and it would be hard to imitate because we'd put some good technology in it.

With hindsight, the prehistory of the company looks very much like a dress rehearsal for a category of device which did not then exist – the mobile phone.

3

Introduction to the Architecture of Symbian OS

3.1 Design Goals and Architecture

Architecture is goal driven. The architecture of a system is the vehicle through which its design goals are realized. Even systems with relatively little formal architecture, such as Unix,[1] evolve according to more or less well-understood principles, to meet more or less well-understood goals. And while not all systems are 'architected', all systems have an architecture.

Symbian OS follows a small number of strong design principles. Many of these principles evolved as responses to the product ethos that was dominant when the system was first being designed.[2] That ethos can be summarized in a few simple rules.

- User data is sacred.
- User time is precious.
- All resources are scarce.

And perhaps this one too, 'while beauty is in the eye of the beholder, elegance springs from deep within a system'.

In Symbian OS, that mantra is taken seriously. What results is a handful of key design principles:

- ubiquitous use of servers: typically, resources are brokered by servers; since the kernel itself is a server, this includes kernel-owned resources represented by R classes

[1] 'Bottom up' and 'informal' typify the Unix design approach, see [Raymond 2004, p. 11].

[2] That is, the ethos which characterized Psion in the early-to-mid 1990s. By then, the company was a leader in the palmtop computer market. It was a product company.

- pervasive asynchronous services: all resources are available to multiple simultaneous clients; in other words, it is a service request and callback model rather than a blocking model
- rigorous separation of user interfaces from services
- rigorous separation of application user interfaces from engines
- engine reuse and openness of engine APIs.

Two further principles follow from specific product requirements:

- pervasive support for instant availability and instant switching of applications
- always-on systems, capable of running forever: robust management and reclaiming of system resources.

Symbian OS certainly aims at unequaled robustness, making strong guarantees about the integrity and safety (security) of user data and the ability of the system to run without failure (to be crash-proof, in other words). From the beginning, it has also aimed to be easy and intuitive to use and fully driven by a graphical user interface (GUI). (The original conception included a full set of integrated applications and an attractive, intuitive and usable GUI; 'charming the user' is an early Symbian OS slogan.[3])

Perhaps as important as anything else, the operating system set out from the beginning to be extensible, providing open application programming interfaces (APIs), including native APIs as well as support for the Visual Basic-like OPL language and Java, and easy access to Software Development Kits (SDKs)[4] and development tools.

However, systems do not stand still; architectures are dynamic and evolve. Symbian OS has been in a state of continuous evolution since it first reached market in late 2000; and for the three years before that it had been evolving from a PDA operating system to one specifically targeting the emerging market for mobile phones equipped with PDA functions. In view of this, it may seem remarkable that the operating system exhibits as much clarity and consistency in design as it does.

---

[3] For example, see almost anything written by David Wood. Today, the GUI is no longer supplied by Symbian, however GUI operation remains intrinsic to the system design. The original integrated applications survive in the form of common application engines across multiple GUIs, although their inclusion is a licensee option.

[4] Symbian no longer directly supplies SDKs, since these are GUI-dependent. Symbian provides significant 'precursor' content to licensees for inclusion in SDKs, including the standard documentation set for Symbian OS APIs.

Architectures evolve partly driven by pressures from within the system and partly they evolve under external pressures, such as pressures from the broad market, from customers and from competition.

Recent major releases of Symbian OS have introduced some radical changes, in particular:

- a real-time kernel, driven by evolving future market needs, in particular, phone vendors chasing new designs (for example, 'single core' phones) and new features (for example, multimedia)
- platform security, driven by broader market needs including operator, user and licensee needs for a secure software platform.

While both are significant (and profound) changes, from a system perspective they have had a relatively small impact on the overall shape of the system. Interestingly, in both cases the pressure to address these particular market needs arose internally in Symbian in anticipation of the future market and ahead of demand from customers.

It is tempting to idealize architecture. In the real world, all software architecture is a combination of principle and expediency, purity and pragmatism. Through the system lifecycle, for anything but the shortest-lived systems, it is part genuine, forward-looking design and part retrofitting; in other words, part architecture and part re-architecture.

Some of the patterns that are present in Symbian OS were also present (or, in any case, had been tried out) in its immediate precursors, the earlier Psion operating systems. The 16-bit operating system (SIBO) had extended the basic server-based, asynchronous, multitasking model of previous Psion products and re-engineered it using object-oriented techniques. SIBO also pioneered the approach to GUI design, designed communications services into the system at a deep level, and experimented with some idioms which have since become strongly identified with Symbian OS (active objects, for example).

In fact, surprisingly many features of Symbian OS have evolved from features of the earlier system:

- the fully integrated application suite: even though Symbian OS no longer includes a user interface or applications, it remains strongly application-centric
- ubiquitous asynchronous services
- optimization for battery-based devices
- optimization for a ROM-based design: unlike other common operating systems, SIBO used strategies such as 'execute-in-place' (XIP) (compare this with MS-DOS, which assumes it is loaded into RAM

to execute) and re-entrancy[5] (MS-DOS is non-re-entrant), as well as a design for devices with only solid-state disks

- sophisticated graphical design: from the beginning, SIBO supported reactive repainting of windows and overlapping windows, in an age of tiled interfaces (for example, Windows 2.0 and the character-mode multitasking user interfaces of the day, such as TopView and DesqView)
- an event-driven programming model
- cross-platform development: the developers' mindset was more that of embedded systems engineering than the standard micro-computer or PC model.[6]

SIBO also introduced some of the programming constraints which show up in Symbian OS, for example forbidding global static variables in DLLs (because the compilers of the day could not support re-entrant DLLs), an early example of using the language and tools to constrain developer choices and enforce design and implementation choices, a consistent theme in Symbian's approach to development.

Symbian OS, or EPOC as it was then, was able to benefit from the experience of the earlier implementation in SIBO. The 16-bit system was, in effect, an advanced prototype for EPOC.

Meanwhile, of course, Symbian OS has continued to evolve. In particular, some crucial market assumptions have changed. Symbian OS no longer includes its own GUI, for example; instead it supplies the framework from which custom, product-ready GUIs such as S60, MOAP and UIQ are built. Hardware assumptions have changed quite radically too. Execute-in-place ROMs, for example, depend on byte-addressable flash silicon (so-called NOR flash); more recently, non-byte-addressable NAND flash has almost wholly superseded NOR flash, making execute-in-place a redundant strategy. Other technology areas, for example display technologies, have evolved almost beyond recognition compared to the 4-bit and 8-bit grayscale displays of earlier times. Not least, the telephony standards that drive the market have evolved significantly since the creation of the first mobile phone networks.

Despite sometimes radical re-invention and change, the original design conception of Symbian OS is remarkably intact.

---

[5] In designing for re-entrant DLLs (that is, re-entrant shared libraries), SIBO was significantly in advance of the available tools. For example, C compilers were poor in this area. Geert Bollen makes the point that it is not just language features that determine whether a given language is suitable for a particular project; the tools infrastructure that supports the language is equally important.

[6] It is interesting to note that Bill Gates has identified as one of Microsoft's key strengths (and, indeed, a key competitive advantage), that it develops all of its systems on its own systems. The advantage breaks down completely in the mobile phone context.

3.2 Basic Design Patterns of Symbian OS

The design principles of a system derive from its design goals and are realized in the concrete design patterns of the system. The key design patterns of Symbian OS include the following:

- the microkernel pattern: kernel responsibilities are reduced to an essential minimum
- the client–server pattern: resources are shared between multiple users, whether system services or applications
- frameworks: design patterns are used at all levels, from applications (plug-ins to the application framework) to device drivers (plug-ins to the kernel-side device-driver framework) and at all levels in between, but especially for hardware adaptation-level interfaces
- the graphical application model: all applications are GUI and only servers have no user interface
- an event-based application model: all user interaction is captured as events that are made available to applications through the event queue
- specific idioms aimed at improving robustness: for example, active objects manage asynchronous services (in preference, for example, to explicit multi-threading) and descriptors are used for type-safe and memory-safe strings
- streams and stores for persistent data storage: the natural 'document' model for applications (although conventional file-based application idioms are supported)
- the class library (the User Library) providing other user services and access to kernel services.

3.3 Why Architecture Matters

'Doing architecture' in a complex commercial context is not easy. Arguably all commercial contexts are complex (certainly they are all different), in which case architecture will never be easy. However, the business model for Symbian OS is particularly complex. While it must be counted as part of Symbian's success, it also creates a unique set of problems to overcome and work around, and to some extent those problems are then manifested as problems for software architecture.

Architecture serves a concrete purpose; it makes management of the system easier or more difficult, in particular:

- managing the functional behavior and supported technologies
- managing the size and performance
- retaining the ability to evolve the system.

Elegance, consistency, and transparency were all early design drivers in the creation of the system. Charles Davies, now Symbian CTO, was the early architect of the higher layers of the operating system.

Charles Davies:

I remember looking at Windows at that time and thinking that this is all very well, here is this Windows API, but just look what's happening underneath it, it was ugly. I wanted something that you could look inside of.

The early 'ethic of robustness', to use his phrase, came straight from the product vision.

Managing the Bounds of the System

In some ways, the hardest thing of all for Symbian is managing the impact of its business model on the properties of the system and, in particular, the problem that Charles Davies calls 'defining the skin' – understanding, maintaining, and managing the bounds of the system under the impact of the business model. As well as originating the requirements push and feeding the requirements pipeline, generating almost all of the external pressure on the system to evolve and grow, licensees and partners also create their own extensions to the system. (S60, arguably, is the most extreme example, constituting a complete system in itself, at around twice the size of the operating system.)

Being clear where to draw the boundary between the responsibilities of Symbian OS and the responsibilities of GUIs, in terms of who makes what and where the results fit into the architecture, becomes difficult. Charles Davies is eloquent on the subject.

Charles Davies:

One of the things I've done since being here is to try and identify where the skin of Symbian OS is, to define the skin. When I was at Psion and we were building a PDA, I understood where the PDA ended and where the things outside the PDA began, and so I knew the boundaries of the product. And then I came to Symbian and Symbian OS, and I thought, where are the boundaries? It's really tough to know where the boundaries are, and I still sometimes wonder if we really know that. That's debilitating from the point of view of knowing what to do. In reality we're trying to fit some kind of rational

> boundary to our throughput, because you can't do everything. We've got, say, 750 people in software engineering working on Symbian OS, and we can't make that 1500 and we don't want to make that 200. So with 750 people, what boundary can we draw that matches a decent product?

In one sense the problem is particular to the business model that Symbian has evolved, and is less a question of pure technology management, which to some extent takes care of itself (or should, with a little help to balance the sometimes competing needs of different licensees), than of driving the operating system vision in the absence of a wider product vision. In that wider sense, the licensees have products; Symbian OS has technologies and it is harder to say what the source of the technology vision for the operating system itself should be. To remain at the front of the field, Symbian OS must lead, but on the whole, customers would prefer that the operating system simply maps the needs of their own leading products. The problem is that by the time the customer product need emerges, the operating system is going to be too late if it cannot already support it. (At least, in the case of complex technologies and, increasingly, all new mobile technologies are complex.) Customers therefore build their own extensions or license them from elsewhere, and the operating system potentially fails under the weight of incompatibilities or missing technologies).

Product companies are easier to manage than technology companies because it is clear what product needs are; either they align with the market needs or the product fails in the market. The Symbian model is harder and forever raises the question of whether Symbian is simply a supplier or integrator to its customers, or an innovator. Is Symbian a product company, where the operating system is the product, or does it merely provide a useful 'bag of bits'?

Architecture is at the heart of the answer. If there is an architecture to describe, then there is more to the operating system than the sum of its parts.

Managing Competitive Threats

There are many external threats to Symbian OS. Some of the threats are household names. Microsoft is an obvious threat, but the likelihood is that Microsoft itself will always be unacceptable to some part of the market, whatever the quality of its technology offering. (It is hard to see Nokia phones, for example, sharing branding with Microsoft Windows, and the same issues no doubt apply to some network operators, but clearly not to all of them.) It is almost as certain that Nokia in turn is unacceptable to some other parts of the market. S60 aims at building a stable of licensees, vendors for whom the advantages of adopting a proven, market-ready

user interface outweigh the possible disadvantages of licensing a solution from a competitor, or the costs of developing an in-house solution. There will always be vendors, though, for whom licensing from Nokia is likely to be unacceptable. Interestingly, the more Microsoft resorts to branding its own phones, in order to increase market share, the more it competes with those it is seeking to license to. It is hard to see any scenario in which the phone market could become as homogeneous as the PC market.

Linux is also a clear and visible threat, even though again there are natural pockets of resistance. Linux, for example, is viral. Linux does not just take out competitors, it takes out whole parts of the software economy, and it is not yet clear what it replaces them with.[7] To put Linux in a phone, for example, seems to require just the same 'old' software economy as to put any other operating system into a phone, dedicated software development divisions which do the same things that other software development divisions do: write code, miss deadlines, fix defects, pay salaries. Linux may be royalty-free, but that translates into 'not-free-at-all' if you have to bring it inside your own dedicated software division. Nonetheless, to ignore Linux would be a (possibly fatal) mistake.

Architecture is part of the answer. If Symbian OS is a better solution, it is because its architecture is more fit for purpose than that of its competitors, not because its implementation is better. Implementation is a second-order property, easy to replace or improve. Architecture, in contrast, is a deep property.

3.4 Symbian OS Layer by Layer

The simplest architectural view of Symbian OS is the layered view given by the Symbian OS System Model.[8]

UI Framework Layer

The topmost layer of Symbian OS, the UI Framework layer provides the frameworks and libraries for constructing a user interface, including the basic class hierarchies for user interface controls and other frameworks and utilities used by user interface components.

The UI Framework layer also includes a number of specialist, graphics-based frameworks which are used by the user interface but which are also available to applications, including the Animation framework, the Front End Processor (FEP) base framework and Grid.

The user interface architecture in Symbian OS is based on a core framework called Uikon and a class hierarchy for user interface controls

[7] Where it is clear, it is not clear how to make a profit from what it replaces them with.

[8] The System Model (see Chapter 5) is relatively constant across different releases, although its details evolve to track the evolution of the architecture.

called the control environment. Together, they provide the framework which defines basic GUI behavior, which is specialized by a concrete GUI implementation (for example, S60, UIQ or MOAP), and the internal plumbing which integrates the GUI with the underlying graphics architecture.

Uikon was originally created as a refactoring of the Eikon user interface library, which was part of the earliest versions of the operating system. Uikon was created to support easier user interface customization, including 'pluggable' look-and-feel modules.

The Application Services Layer

The Application Services layer provides support independent of the user interface for applications on Symbian OS. These services divide into three broad groupings:

- system-level services used by all applications, for example the Application Architecture or Text Handling
- services that support generic types of application and application-like services, for example personal productivity applications (vCard and vCal, Alarm Server) and data synchronization services (OMA Data Sync, for example); also included are a number of key application engines which are used and extended by licensees (Calendar and Agenda Model), as well as legacy engines which licensees may choose to retain (Data Engine)
- services based on more generic but application-centric technologies, for example mail, messaging and browsing (Messaging Store, MIME Recognition Framework, HTTP Transport Framework).

Applications in Symbian OS broadly follow the classic object-oriented Model–Viewer–Controller (MVC) pattern. The framework level support encapsulates the essential relationships between the main application classes (representing the application data model, the views onto it, and the document and document user interface that allow it to be manipulated and persisted) and abstracts all of the necessary underlying system-level behavior. In principle, a complete application can be written without any further direct dependencies (with the exception of the User Library).

The Application Services layer reflects the way that the system as a whole has evolved. On the one hand, it contains essential application engines that almost no device can do without (the Contacts Model for example), as well as a small number of application engines that are mostly now considered legacy (e.g. the WYSIWYG printing services and the office application engines, including Sheet Engine, a full spreadsheet engine more appropriate for PDA-style devices). On the other hand, it contains (from Symbian OS v9.3) the SIP Framework, which provides the foundation for the next generation of mobile applications and services.

Java ME

In some senses, Java does not fit neatly into the layered operating system model. Symbian's Java implementation is based around:

- a virtual machine (VM) and layered support for the Java system which complements it, based on the MIDP 2.0 Profile
- a set of standard MIDP 2.0 Packages
- an implementation of the CLDC 1.1 language, I/O, and utilities services
- a number of low-level plug-ins which implement the interface between CLDC, the supported packages, and the native system.

Java support has been included in Symbian OS from the beginning, but the early Java system was based on pJava and JavaPhone. A standard system based on Java ME first appeared in Symbian OS v7.0s. Since Symbian OS v8, the Java VM has been a port of Sun's CLDC HI.

The OS Services Layer

The OS Services layer is, in effect, the 'middleware' layer of Symbian OS, providing the servers, frameworks, and libraries that extend the bare system below it into a complete operating system.

The services are divided into four major blocks, by broad functional area:

- generic operating system services
- communications services
- multimedia and graphics services
- connectivity services.

Together, these provide technology-specific but application-independent services in the operating system. In particular, the following servers are found here:

- communications framework: the Comms Root Server and ESock (Sockets) Server provide the foundation for all communications services
- telephony: ETel (Telephony) Server, Fax Server and the principal servers for all telephony-based services
- networking: the TCP/IPv4/v6 networking stack implementation
- serial communications: the C32 (Serial) Server, providing standard serial communications support

- graphics and event handling: the Window Server and Font and Bitmap Server provide all screen-drawing and font support, as well as system- and application-event handling
- connectivity: the Software Install Server, Remote File Server and Secure Backup Socket Server provide the foundation for connectivity services
- generic: the Task Scheduler provides scheduled task launching.

Among the other important frameworks and libraries found in this layer is the Multimedia Framework (providing framework support for cameras, still- and moving-image recording, replay and manipulation, and audio players) and the C Standard Library, an important support library for software porting.

The Base Services Layer

The foundational layer of Symbian OS, the Base Services layer provides the lowest level of user-side services. In particular, the Base Services layer includes the File Server and the User Library. The microkernel architecture of Symbian OS places them outside the kernel in user space. (This is in contrast to monolithic system architectures, such as both Linux and Microsoft Windows, in which file system services and User Library equivalents are provided as kernel services.)

Other important system frameworks provided by this layer include the ECom Plug-in Framework, which implements the standard management interface used by all Symbian OS framework plug-ins; Store, which provides the persistence model; the Central Repository, the DBMS framework; and the Cryptography Library.

The Base Services layer also includes the additional components which are needed to create a fully functioning base port without requiring any further high-level services: the Text Window Server and the Text Shell.

The Kernel Services and Hardware Interface Layer

The lowest layer of Symbian OS, the Kernel Services and Hardware Interface layer contains the operating system kernel itself, and the supporting components which abstract the interfaces to the underlying hardware, including logical and physical device drivers and 'variant support', which implements pre-packaged support for the standard, supported platforms (including the Emulator and reference hardware boards).

In releases up to Symbian OS v8, the kernel was the EKA1 (Kernel Architecture 1) kernel, the original Symbian OS kernel. In Symbian OS v8, the EKA2 (Kernel Architecture 2) real-time kernel shipped for the first time as an option. (It was designated Symbian OS v8.1b; Symbian OS v8.1a is

the Symbian OS v8.1 release with the original kernel architecture.) From Symbian OS v9, EKA1 no longer ships and all systems are based on the real-time EKA2 kernel.[9]

3.5 The Key Design Patterns

Probably the most pervasive architectural pattern in Symbian OS is the structuring client–server relationship between collaborating parts of the system. Clients wanting services request them from servers, which own and share all system resources between their clients.

Another widely used pattern is the use of asynchronous methods in client–server communications. Together, these two patterns impose their shape on the system. Like any good architecture, the patterns repeat at multiple levels of abstraction and in all corners of the system.

A third pervasive pattern is the use of a framework plug-in model to structure the internal relationships within complex parts of the system, to enable flexibility and extensibility. Flexibility in this context means run-time flexibility and is particularly important when resources are constrained. The ability to load the requested functionality on demand enables more efficient use of constrained resources (objects which are not used are not created and loaded). Extensibility is important too in a broader sense. The use of plug-ins enables the addition of behavior over a longer timescale without re-architecting or re-engineering the basic design. An example is the structure of the telephony system which encapsulates generic phone concepts which are then extended, for example for GSM- or CDMA-specific behaviors, by extension frameworks. The use of plug-ins also enables licensees to limit or extend functionality by removing or replacing plug-in implementations.

At a lower level, Symbian OS makes much use of specific, local idioms. For example, active objects are the design idiom which makes asynchronicity easy and are widely used. ('Asynchronicity' here means the ability to issue a service request without having to wait for the result before the thread of execution can continue.) Encapsulating asynchronicity into active objects is an elegant object-oriented design. (Active objects are examples of cooperative multitasking: multiple active objects execute in effect within the context of a single thread. Explicit multithreading is an example of non-cooperative multitasking, that allows pre-emption.)

Symbian OS has also evolved a number of implementation patterns, including 'leaving' functions and the cleanup stack, descriptors for safe strings, local class and member naming conventions and the use of manifest constants for some basic types.

[9] This history is described in detail in [Sales 2005], the in-depth, authoritative reference.

Symbian's microkernel design dates back to its original conception, but becomes even more significant in the context of the new real-time kernel architecture. The real-time architecture is essential for a system implementing a telephony stack, which depends on critical timing issues, and is also becoming increasingly important for fast, complex multimedia functionality. Together, phone and multimedia are arguably the most fundamental drivers for any contemporary operating system. As mobile phones, in particular, reach new levels of multimedia capability, to become fully functional converged multimedia devices (supporting streamed and broadcast images and sound, e.g. music streaming, two-way streaming for video phone conferencing and interactive broadcast TV), achieving true real-time performance has become an essential requirement for a phone operating system. The real-time kernel allows Symbian OS to meet that requirement, making it a suitable candidate for directly hosting a 3G telephony stack.

The real-time kernel architecture also introduces important changes (in particular to mechanisms such as interprocess communication) to support the new platform security model introduced from Symbian OS v9. (Strictly speaking, the security model is present in Symbian OS v8 but implements a null policy. The full security model, which depends on the new kernel architecture, is present from Symbian OS v9.)

The Client–Server Model

In Symbian OS, all system resources are managed by servers. The kernel itself is a server whose task is to manage the lowest level machine resources, CPU cycles and memory.

From the kernel up, this pattern is ubiquitous. For example, the display is a resource managed by the Window Server; display fonts and bitmaps are managed by the Font and Bitmap Server; the data communications hardware is managed by the Serial Server; the telephony stack and associated hardware by the Telephony Server; and so on all the way to the user-interface level, where the generic Uikon server (as specialized by the production GUI running on the final system) manages the GUI abstractions on behalf of application clients.

Threads and Processes

The client–server model interacts with the process and threading model in Symbian OS. While this is in keeping with a full object-oriented approach, which objectifies machine resources in order to make them the fundamental objects in the system, it can also cause confusion.

In Symbian OS, threads and processes are defined in [Sales 2005, Chapter 3] as follows:

- threads are the units of execution which the kernel scheduler schedules and runs
- processes are collections of at least one but possibly multiple threads which share the same memory address space (that is, an address mapping of virtual to physical memory).

Processes in other words are units of memory protection. In particular each process has its own heap, which is shared by all threads within the process. (Each thread has its own stack.)

A process is created as an instantiation of an executable image file (of type EXE in Symbian OS) and contains one thread. Creation of additional threads is under programmer control. Other executable code (for example, dynamically loaded code from a DLL file) is normally loaded into a dynamic-code segment attached to an existing process. Loading a DLL thus attaches dynamic code to the process context of the executing thread that invokes it.

Each server typically runs in its own process,[10] and its clients run in their own separate processes. Clients communicate with the server across the process boundary using the standard client–server conventions for interprocess communication (IPC).[11]

As Peter Jackson comments, Symbian OS falls somewhere between conventional operating system models in its thread and process model.

Peter Jackson:

Most of the threads versus processes issues are to do with overhead. In some operating systems, processes are fairly lightweight, so it's very easy to spawn another process to do some function and return data into a common pool somewhere. Where the process model is more heavyweight and the overhead of spawning another one is too great, then you invent threads and you let them inherit the rest of the process, so the thread is basically just a way of scheduling CPU activity. In Symbian OS, you can use whichever mechanism is appropriate to the requirements.

Server-Side and Client-Side Operations

Typically a server is built as an EXE executable that implements the server-side classes and a client-side DLL that implements the client-side interface to the server. When a client (either an application or another

[10] There are some exceptions for reasons of raw speed.

[11] [Sales 2005] defines the Symbian OS client–server model as inter-thread communication (ITC), which is strictly more accurate than referring to interprocess communication (IPC). However, arguably the significance of client–server communications is the crossing of the process boundary.

system service) requests the service, the client-side DLL is attached to the calling process and the server-side executable is loaded into a new dedicated process (if it is not already running).

Servers are thus protected from their clients, so that a misbehaving client cannot cause the server to fail. (The server and client memory spaces are quite separate.) A server has responsibility for cleaning up after a misbehaving client, to ensure that resource handles are not orphaned if the client fails.

At the heart of the client–server pattern therefore is the IPC mechanism and protocol, based on message passing, which allows the client in its process, running the client-side DLL, to communicate via a session with the server process. The base classes from which servers and their client-side interfaces are derived encapsulate the IPC mechanisms.

The general principles are as follows:[12]

- The client-side implementation, running in the client process, manages all the communications across the process boundary (in the typical case) with the server-side implementation running in the server process.
- The calling client connects to the client-side implementation and creates a session, implemented as a communications channel and protocol created by the kernel on behalf of the server and client.
- Client sessions are typically created by calling `Connect()` and are closed using `Close()` methods, in the client-side API. The client-side calls invoke the standard client–server protocol methods, for example `RSessionBase::CreateSession()` and `RProcess::Create()`. On a running server, this results in the client session being created; if the server is not already running, it causes the server to be started and the session to be created.
- The client typically invokes subsessions that encapsulate the detailed requests of the server-defined protocol. (In effect, each client–server message can be thought of as creating a subsession.)
- Typically, client-side implementations derive from `RSessionBase`, used to create sessions and send messages.
- Typically, the server side derives from `CServer`.

Servers are fundamental to the design of Symbian OS, and are (as the mantra has it) the essential mechanism for serializing access to shared resources, including physical hardware, so that they can be shared by multiple clients.

[12] The best description is [Stichbury 2005, Chapter 12].

Andrew Thoelke:

It's not so much that there is a server layer in the operating system as a hierarchy. It's very much a hierarchy and there are a lot of shared services. Some of them are shared by quite a few components and some of them really support just a very small part of the system, and of course those shared services may build on top of one or more client–server systems already.

Client–server is a deep pattern that is used as a structuring principle throughout the system.

Asynchronous Services

Another deep pattern in the system is the design of services to be asynchronous.

System responsiveness in a multitasking system (the impression that applications respond instantly and switch instantly) depends on asynchronous behavior; applications don't wait to finish processing one action before they are able to handle another.

The alternatives are blocking, or polling, or a combination of both. In a blocking request (the classic Unix pattern), the calling program makes a system call and waits for the call to return before continuing its processing. Polling executes a tight loop in which the caller checks to see if the event it wants is available and handles it when it is. (Polling is used by MS-DOS, for example, to fetch keystrokes from the keyboard.)

Blocking is unsatisfactory because it blocks others from accessing the system call which is being waited on, while it is waiting. Polling is unsatisfactory because code which is functionally idle, waiting for an event, is in reality not idle at all, but continuously executing its tight polling loop.

Blocking reduces responsiveness. Polling wastes clock cycles, which on a small system translates directly to power consumption and battery life.

Charles Davies:

Asynchronous services was driven by battery life. We were totally focused on that. For example on one of the Psion devices, we stopped the processor clock when it was idle. I don't know if that was innovative at the time. We certainly didn't copy it from anybody else, but we had a static processor. Usually in an idle process, the operating system is doing an idle loop. But we didn't do that, we stopped the clock on the processor and we turned the screen off, and that was fundamental to the design.

Typically, client–server interactions are asynchronous.

The Plug-in Framework Model

A final high-level design pattern, the plug-in framework model is used pervasively in Symbian OS, at all levels of the system from the UI Framework at the top to the lowest levels of hardware abstraction at the bottom.

A framework (as its name suggests) is an enclosing structure. A plug-in is an independent component that fits into the framework. The framework has no dependency on the plug-in, which implements an interface defined by the framework; the plug-in has a direct, but dynamic, dependency on the framework.

Frameworks are one of the earliest design patterns (going back to the time before design patterns were called design patterns, in fact) [Johnson 1998]. While, in principle, nothing limits them to object-oriented design, they lend themselves so naturally to object-oriented style that the two are strongly identified. A key principle of good design (again, not limited to object-oriented design but closely identified with it) is the separation of interface from implementation. On a small scale, this is what designing with classes achieves: a class abstracts an interface and its expected behavior and encapsulates its implementation. Frameworks provide a mechanism for this kind of abstraction and encapsulation at a higher level. As is often said, frameworks enable a complete design to be abstracted and reused.[13] Frameworks are therefore a profound and powerful way of constructing an object-oriented system.

In detail, a framework in Symbian OS defines an external interface to some part of the system (a complete and bounded logical or functional part) and an internal plug-in interface to which implementers of the framework functionality (the plug-ins) conform. In effect, the framework is a layer between a calling client and an implementation. In the extreme case, a 'thin' framework does little more than translate between the two interfaces and provide the mechanism for the framework to find and load its plug-ins. A 'thicker' framework may do much more, providing plug-in interfaces which are highly abstracted from the external visible client interface. Symbian OS contains frameworks at both extremes and most points in between.

Because in Symbian OS a framework exposes an external interface to a complete, logical piece of the system, most frameworks are also implemented as servers.

As well as providing interface abstraction and separation from implementation, and flexibility through decoupling, frameworks also provide a natural model for functional extension. This approach is used for example by the telephony-server framework to provide an open-ended design. The core framework supports generic telephony functionality based around a small number of generic concepts. Framework extensions implement

[13] A framework is 'reusable design' as [Johnson 1998] puts it.

the specialized behaviors which differentiate landline from mobile telephony, data from voice, circuit- from packet-switched, GSM from CDMA, and so on.

As well as this 'horizontal' extension of the range of functionality of the framework, such a plug-in also defines the interfaces which are implemented 'vertically' by further plug-ins that provide the actual services.

Because the plug-in framework model is pervasive, Symbian OS provides a plug-in interface framework. (Available since Symbian OS v7.0s but universally enforced since Symbian OS v8.0 as part of the phased introduction of Platform Security.) The plug-in framework (also known as ECom) standardizes the mechanisms and protocols that allow frameworks to locate and load the plug-ins which provide their implementations, and for plug-ins to register their presence and availability in the system as implementation modules.

Clearly, plug-ins pose a potential security threat because they provide a mechanism for untrusted (that is, externally supplied) code to be loaded into the processes of some system components (although the microkernel architecture keeps them well away from the kernel). The plug-in framework therefore enforces the security model on plug-ins before they are loaded [Heath 2006].

Another area in which plug-ins pose potential risks to the system is in performance. Potentially, a badly designed or poorly implemented plug-in can damage the performance of the framework that loads it. The plug-in model can also make it hard to understand the dynamic behavior of the operating system and, in particular, can make system-level debugging tricky, since the system can become (from the perspective of the debugger) highly indeterministic, unpredictable and unreproduceable.

However, enabling a pervasive model of run-time rather than static loading can boost system performance. Plug-ins are loaded on request; if they are not requested, they are not loaded, saving loading time and system resources (including RAM, on systems that do not provide execute-in-place).

An interesting example of just how pervasive the plug-in framework pattern is in Symbian OS is the original implementation of applications as plug-ins to the application and UI Framework rather than as more conventional executables. (This architecture changes somewhat in Symbian OS v9, where applications are implemented as EXEs rather than DLLs, while retaining other characteristics of plug-ins.)

In implementation terms, an ECom plug-in is implemented as a polymorphic DLL and a resource (RSC) file. The DLL entry point is a factory function that instantiates the plug-in object. All system plug-ins are stored into well-known locations, as required by the security model.

The plug-in framework provides a standard and universal mechanism for binding implementations (plug-ins) to interfaces (frameworks) at run

time, together with the mechanisms for packaging multiple interface implementations into a single DLL (that is, loading multiple implementations at once, to improve performance), plug-in registration and implementation versioning, discovery and loading including boot-time discovery optimizations to avoid run-time overhead, and cleanup after unloading plug-ins. (A plug-in instance cannot destroy itself, because its destructor code would be part of the code being removed from memory.) The framework also provides security-policy definition and policing mechanisms.

The plug-in framework is implemented as a server, in effect a broker between frameworks and conforming plug-ins, managing those plug-ins as a resource to its framework clients.

Microkernel Architecture

Symbian OS has a microkernel architecture, which sets it apart from operating systems such as Microsoft Windows and Linux.[14] In Symbian OS, core services that would be inside the kernel in a monolithic operating system are moved outside. The pervasive use of the client–server architecture, and the protection of system code from clients which follows from it, guarantees both the robustness and high availability of these services. The goal is a robust system that is also responsive and extensible; experience suggests that the design achieves it.

Andrew Thoelke:

The actual client–server architecture, the division into processes across the operating system and the boundary of the kernel, means that the actual privileged mode software is much smaller than in desktop operating systems. It's very nearly theoretical microkernel, but not completely truly microkernel because device drivers all run kernel side, and a true microkernel would say that device drivers should run user side, and who knows maybe we'll get there in a few years time. But all file system services, all higher level comms services including networking, and the windowing software for example, all run user side.

If anything the new EKA2 kernel architecture goes beyond the microkernel design and encapsulates the most fundamental kernel primitives within a true real-time nanokernel, supporting an extended kernel that implements the remaining Symbian OS kernel abstractions, but is equally

[14] There are microkernel implementations of Unix, based on the Mach microkernel. Mac OS X is an example; it is built as a Berkeley Unix variant with a Mach microkernel and proprietary user interface layer. Other microkernel designs include QNX, which is an operating system similar to Unix, but not Unix; Chorus, which is not just a microkernel but also object-oriented and which, like Mach, is capable of hosting Unix; and iTron, which is an important mobile-phone operating system in Japan.

capable of supporting 'personality' layers to mimic the interface of any other operating system. But the essential elegance of the Symbian OS kernel design goes right back to its earliest days.

Martin Tasker:

The Symbian model is that you're either a user thread or a kernel thread, and if you're a user thread then either you're an application thread, which has a session with the window server and interacts with the user, or you're a server thread which has no interaction with the user. And if you're a server thread, well then you sit around waiting for client requests to happen and when they do you service them, and in fact the kernel has a server and it does just that. There are a couple of kernel calls which are handled by something known as fast execs, which don't involve the kernel server. But the design philosophy of the kernel is to make those things very short and sweet and to put most of the work into the server. I think that's a cool architecture. Some of it goes down to Colly Myers's explainability requirement, that it takes more than an average programmer to implement any of this stuff, but any average programmer should be able to use it.

The lineage of course can be traced back to the precursor Psion systems.

Andrew Thoelke:

It owes its design very much to the heritage of Series 3. Colly Myers took that same OS structure, that you've got a small amount of protected mode software that can do everything, and that even all the file system and file services actually operate in a separate process from that and have less privileges, and that you have a very tightly integrated client–server architecture that actually binds everything together. That is definitely quite different to what you see in a lot of other systems.

Notwithstanding the move to the EKA2 kernel architecture, at a high level the lineage is still visibly present.

Martin Tasker:

The change from EKA1 to EKA2 is a hugely significant change. But at the system-design level, you know that change hasn't actually radically altered the system design at all. It's still either application processes or server processes, and that design was actually pioneered all the way back in SIBO, and it hasn't changed much since then, and the reason is: it's a proven design.

3.6 The Application Perspective

Symbian OS has been designed above all to be an application platform (although it might be argued that that has begun to change, and that in the latest devices it has become primarily an engine for driving fast, mobile data communications). Applications have always been an essential part of the system. The early operating system shipped with a complete set of productivity and communications applications targeting connected PDAs. Although Symbian OS no longer supplies a GUI and user-ready applications but only common application engines, Symbian OS phones now ship with more built-in applications than ever before, supplied either with the licensee GUI or as extras provided by the phone vendor or network operator.

Charles Davies:

Symbian started off as an operating system plus an application suite. We never designed it as an operating system independently of the suite of applications.

Just as importantly, both S60 and UIQ are also explicitly pitched as open platforms for third-party applications and provide extensive support for developers including freely available SDKs, support forums and tools.

From the beginning the approach to applications has been graphics-based. Like much else, the approach has evolved and, in particular, it has evolved as Symbian's user interface strategy has evolved. However, the principles of application structure have been essentially mature since the first release of S60 and UIQ in 2002.

Uikon is the topmost layer of Symbian OS. It provides the framework support on which a production user interface is built. The three currently available custom user interfaces are S60, UIQ and MOAP, but there is no engineering reason why any licensee should not build its own bespoke user interface, which indeed is precisely the origin of S60 and MOAP. Uikon abstracts application and control base classes in the Application Architecture and Control Environment class hierarchies to create generic GUI application classes (that is, classes free of a look and feel policy) which are customized by the custom user interface. The custom user interface abstracts the Uikon policy-free base classes to provide the policy-rich classes that applications derive from.

Uikon thus integrates the underlying support of the Application Architecture and the Control Environment to create a framework from which (as abstracted by the custom user interface), applications derive. Uikon is a framework and applications behave recognizably as plug-ins. Uikon is implemented as a server.

The Structure of an Application

Every application is built from three basic classes:[15]

- an application class, derived from the Application Architecture (`CApaApplication`)
- a document class, derived from the Application Architecture (`CEikDocument`)
- an application user interface class, derived from the Control Environment (`CCoeAppUiBase`).

These classes provide the fundamental application behavior. However, two important parts of the application are missing from this structure: the application view, which is the screen canvas the application uses to display its content, and the application data model and data interface implementations, which encapsulate the application 'engine'.

The classic application structure expects that the data model (the data-oriented application functionality) exists independently of the GUI implementation of the application and is, therefore, independent of any user interface classes. It is hooked into the user interface by a member pointer (`iModel`) in the document class. The classes specific to the user interface then interact with it purely through the APIs it exposes.[16]

Charles Davies:

We always had that structuring of applications, the idea of separating the UI from the application engine. That was an early design principle and it was the design guidance for application writers. We knew about Model–View–Controller, and we thought of an application engine as a model, and our design guidance was to keep the application logic separate from the UI. Not because we anticipated at that time multiple UI flavors, but because we recognized something more fundamental in terms of writing an application. That you might write an application and decide to improve the design of the UI, where the refinement of the UI was just pragmatic, the basic functional application logic stayed the same. So if you could separate those two things, that was good, and that led to the terminology of application engines.

[15] This is the 'classic' application structure, with roots in the Eikon applications of Psion Series 5. Both UIQ and S60 extend the design patterns for applications. See [Edwards 2004, p. 184] for discussions of the 'dialog-based' and 'view-switching' S60 application structure. UIQ applications also extend the basic pattern with custom view classes.

[16] This is in fact a very powerful design principle, implying, for example, that the data model can run without a direct user interface at all. Engines designed this way are independently testable and intrinsically highly portable between different user interfaces. The principle runs deep in the Symbian ethos, as witnessed by the presence of engines independent of the user interface in the operating system itself.

In Symbian OS, a control is a drawable screen region (in other words, the owner of screen real estate). The Application view class is derived directly from the Control Environment control base classes.

On small devices, where screen real estate is scarce, desktop-style windowing is not appropriate. A more natural approach for small displays is to switch whole-screen views, for example switching between a list-style view of contact names and a record-style view of the details of a single contact. Applications therefore typically define a hierarchy of views, with the main application view at the root.

Because Symbian OS is multitasking, multiple applications can be running at once, even though only one (the foreground application) will be presenting its view on the display. Both S60 and UIQ support switching directly between views in different applications, including launching the view of a new application inside the context of the current one (for example selecting a phone number from within a Contact entry and immediately switching to the phone application and dialing the number).

Symbian's application structure makes much of the detail of the application user interface programmable solely via resource files. Resource files are compiled separately as part of the application build process and linked into the built application, providing a natural mechanism for language localization (all text strings used within an application can be isolated in resource files and recompiled to a new language without having to recompile the application). Resource files are also compressed.

Charles Davies:

We lived in tougher times as far as Moore's law was concerned in those days. Resource files were around in contemporary GUI systems at that time. But from the beginning we did Huffman compression on resource files, and we were careful about the amount of information we put in them.

Uikon

The most striking fact about Symbian OS at the user interface level is its support for a replaceable user interface, and indeed the fact that it ships without a native user interface at all. (User-interface-dependent components are shipped only with a TechView test user interface.)

While it seems fair to say that Symbian did not get its user interface strategy right first time (in particular, the Device Family Reference Design (DFRD) strategy looks, with hindsight, to have been naïve), nonetheless the operating system has been able to support multiple licensees, each having a distinct user-interface philosophy, occupying different positions in the market and spanning diverse geographical locations. Those differences are encapsulated in the differences between the user interfaces that have evolved for Symbian OS.

S60 builds on the classic Nokia user interface to provide a simple, key-driven but graphically rich and arresting user interface. In contrast, UIQ is firmly pen-based and targets high-end phones with rich PDA-like functionality including pen-based handwriting recognition. MOAP aims squarely at its solely Japanese market, providing a graphically busy user interface featuring Kanji as well as Roman text and animated cartoon-style icons.

File System or 'Object Soup' Storage Model

FAT is the 'quick and dirty' file system that MS-DOS made famous. When work on EPOC started, the Apple Newton was a leading example of a different way to approach consumer computing (different, for example, from the MS-DOS-based Hewlett Packard machines which were the leading competitor for Psion's Series 3). Instead of a conventional file system the Newton employed an 'object soup' storage model.[17]

On any useful system, data requires a lifetime beyond that of the immediate context in which it is created, whether that means storing system settings, saving the memo you have just written to a file, or storing the contact details you have just updated.

Charles Davies:

We had a normal file system on the Series 3. When we went to C++, we talked a lot about persistent models of object-oriented programming, and we went for stream storage. We narrowly rejected SQL in favor of stream storage. I remember the design ideas around at the time, and it was done in the interests of efficiency. Different applications were having to save the same system objects and we were having to duplicate that code. So for something like page margins, which was a system structure, if that object knew how to serialize itself, that would solve the problem. You do that by having serialization within the object, so objects that might reasonably want to be persisted could persist themselves. And that was in the air, I mean Newton had its soup at that time which I think was object-oriented, and there was a belief at that time that object-oriented databases were it, and that objects ought to be seen as something that existed beyond the lifetimes of processes.

Objects, in other words, can be viewed as more than just the run-time realizations of object-oriented code constructs. However, in terms of the standards of the day, approaches based on something other than a file system were certainly the exception. The big challenge in maintaining data is that of data format and compatibility, ensuring that the data remains accessible. Any device which aims to be interoperable

[17] 'Object soup' is described in [Hildebrand 1994].

(in any sense) with other devices faces a similar challenge. In both cases, the design is immediately constrained in how far it can deviate from the data-format conventions of the day. For EPOC at that time, compatibility with desktop PCs was an essential requirement. For Symbian OS now, the requirement is more generalized to compatibility with other devices of all kinds. Probably the most important test case for both is readability of removable media file systems. (All other cases in which a Symbian OS device interoperates with another device can be managed by supporting communications protocols and standard data formats, which are independent of the underlying storage implementation.)

While external compatibility does not determine internal data formats, the need to support FAT on removable cards probably tipped the balance towards an internal FAT filing system. One (possibly apocryphal) story has it that the decision to go with FAT was a Monday morning *fait accompli* after Colly Myers had spent a weekend implementing it.

Peter Jackson:

There were periods when we explored all sorts of quite radical ideas but in the end we always came back to something fairly conservative, because if you take risks in more than one dimension at a time it doesn't work. So I spent quite a lot of time at one stage investigating an object-oriented filing system. But one day I think Colly Myers had a sudden realization and he just said, 'Let's do FAT', and he was probably right.

But FAT is not the whole story. In fact, Symbian OS layers a true object-oriented persistence model on top of the underlying FAT file system. As well as a POSIX-style interface to FAT, the operating system also provides an alternative streaming model.

It is an interesting fact that data formats, whether those of MS-Word or Lotus 1-2-3 or MS-Excel, have proved to be powerful weapons in the marketplace, in some cases almost more so than the applications which originated them. (The Lotus 1-2-3 data format lives on long after the demise of the program and, indeed, of the company.) Data in this sense is more important than the applications or even the operating systems with which it is created.

Peter Jackson:

The layout of the file is an example of a binary interface and, as software evolves, typically those layouts change, sometimes in quite an unstructured or unexpected way, because people don't think of them as being a binary interface that you have to protect. So the alternative way of looking at things is to say you don't think about that, you ignore the layout of the file. What you

do is you look at the APIs, and you program all your file manipulation stuff to use the same engines that originated the data in the first place.

In effect, this is the approach that Symbian adopted. But it has a cost.

Charles Davies:

We went for an architecture in which applications lost control of their persistent data formats, and in retrospect I think that was a mistake, because data lasts longer than applications. The persistence model is based on the in-memory aggregation in the heap of whatever data structure you're working with. For example, if it's a Contacts entry, then it consists of elements and you stream the elements. One problem is that if you try to debug it and you're looking at a file dump, its unfathomable. It's binary, it's compressed, so it's very efficient in the sense that when you invent a class it knows how to stream itself, so it's a sort of self-organizing persistence model, but the data dump is unfathomable. The second problem is that when you change your classes it changes how they serialize. So it works. But if you add a member function which needs to be persisted, then you change the data format. You lose data independence, and that stops complementers from working with your formats too. So we sacrificed data independence. And because that data has to carry forward for different versions of the operating system, you get stuck with that data format and you end up with a data migration problem. So I think that was a mistake. It would have been worth it to define data-independent formats. In my view that's what XML has proved, the XML movement has shown that data sticks longer than code.

In some ways, implementing a persistence model on top of a FAT system leads to the worst of both worlds, on the one hand missing out on the benefits of MS-DOS-style data independence, and on the other missing out on Newton-style simplicity.

Peter Jackson:

If you implement your permanent store structure in terms of a database design then you have all the advantages of being able to use database schema idioms to talk about what you're doing, and it turns out that those idioms now are fairly stable and universal. So I think there are examples where we have pruned away the databaseness of an application because we thought our customers didn't really want a database – but that may be a bad thing if one day our customers decide they want more than just flat data.

Store and DBMS

The native persistence model is provided by Store, which defines Stream and Store abstractions. Together they provide a simple and fully object-oriented mechanism for persistence:

- A Stream is an abstract interface that defines `Externalize()` and `Internalize()` methods for converting to and from internal and external data representations, including encrypted formats.
- A Store is an abstract interface that defines `Store()` and `Restore()` methods for persisting structured collections of streams, which represent whole documents. Store also defines a dictionary interface which allows streams to be located inside a store.

Symbian OS also includes DBMS, a generic relational database API layered on top of Store, as well as implementations including a lightweight, single-client version (for example, for use by a single application that wants a database-style data model which will not be shared with others). Databases are stored physically as files (single client databases may also be stored in streams).

Database queries are supported either through an SQL subset or a native API. Since the introduction of platform security, the DBMS implementation supports an access-policy mechanism to protect database contents.

3.7 Symbian OS Idioms

C++ is the native language of Symbian OS. Symbian's native APIs therefore are C++ APIs (although API bindings exist for other languages: OPL, Java and, most recently, Python). C++ is a complex, large and powerful language. The way C++ is used in Symbian OS is often criticized for being non-standard. For example, the Standard Template Library (STL) is not supported, the Standard Library implementation is incomplete, and POSIX semantics are only partly supported. Since Symbian OS competes with systems which do support standard C++, there is also little doubt that the operating system will evolve towards supporting more standard C++. But, like it or not, true native programming in C++ on Symbian OS requires understanding and using its native C++ idioms.

Among some developers inside the company the view has been unashamedly one of, 'Those who can, will; those who can't should use Java, Python, or even OPL'.[18] While that may not make for mass market appeal for Symbian C++ itself, the fact is that programming on

[18] For example, see the remarks by David Wood in Chapter 18.

any platform requires specialist expertise as well as general expertise, and, in that, Symbian OS is no different. The skill level required is commensurate with the programming problem. It is far from easy to write software for consumer devices on which software failures, glitches, freezes and crashes – things people put up with regularly on their PCs – are simply not an option. Mobility, footprint, battery power, the different user expectations, screen size, key size and all the other specifics of their small form factors make mobile devices not at all like desktop ones; phones, cameras, music players and other consumer devices are different.

Symbian OS idioms are not casual idiosyncrasies; they are deliberate constraints on the C++ language devised to constrain developer choices, consequences of the market the operating system targets, and of the embedded-systems nature of ROM-based devices. Strictly speaking, they are less architectural than implementational but, in terms of the overall design, they are important and they have an important place in the history of the evolution of the system. Understanding them is essential to understanding what is different about Symbian OS, and what is different about mobile devices. There are some large-scale differences.

- Lack of a native user interface means that the development experience is significantly different for device creation developers using the TechView test user interface than for developers later in the product lifecycle using S60, UIQ or MOAP.
- The build system is designed for embedded-style cross-compilation, which is a different experience from desktop development.
- Idioms have evolved to support the use of re-entrant, ROM-based DLLs, for example disallowing global static data.
- Other optimizations for memory-constrained, ROM-based systems result in some specific DLL idioms (link by ordinal not name, for example).

There are what might be described as language-motivated idioms:

- descriptors
- leaving functions
- the cleanup stack
- two-phase construction.

And there are some design-choice idioms:

- active objects and the process and threading model
- UIDs

- static libraries and object-oriented encapsulation
- resource files to isolate locale-specific data, for example, text strings.

Active Objects

Active objects are an abstraction of asynchronous requests and are designed to provide a transparent and simple multitasking model.

An active object is an event handler which implements the abstract interface defined by the `CActive` class and consists of request and cancellation methods, which request (or cancel) the service the object should handle, and a `Run()` method which implements the actual event handling. When the requested service completes and there is a result to be handled, a local active scheduler invokes the active object's `Run()` method to handle the completed event.

An active scheduler is created by the UI Framework for each application. All active objects invoked by an application (but only that application's active objects) share a single thread, in which they are not pre-empted (i.e. they are scheduled in priority order by the scheduler).

Active objects are a pervasive Symbian idiom and provide a non-pre-emptive multitasking alternative to explicitly creating multithreaded programs (although that option remains available to developers), as a solution to the problem of managing multiple paths of execution within a program, in the context of an event-based, reactive application model. From the perspective of a GUI application developer they offer a much easier solution than multithreading, in effect handing off the awkward details to the system.

Charles Davies:

Our model for events was very much asynchronous events and signals and requests. So what we had first of all, and it's what other systems have too, is that you make one or more requests for events, and events include timers and serial events and all kind of events that can come out of anywhere, not just user-originated events. So you just set off a large number of events and then you wait for any one of them to come through. So things need to be able to respond to events from multiple sources. Now Windows had a way of handling this. There's a Windows API, though it's not very elegant. The problem is, it's tied to the GUI programming model. In Windows you have to run up the whole GUI to get the event model going, and we thought that was a real weakness in mobile devices. We thought that servers needed this as well, that servers sit there waiting for events from multiple sources, events like 'my client has died', which comes from a different source than the message channel saying 'here's the next request from the client'.

The event-driven model is essentially a state-machine model. But, except within niche areas such as communications programming, these

were not widely used patterns, especially for applications programming. And except for those familiar with Windows at the time, or with other GUI systems such as Amiga and Macintosh, the event-driven application model was not widely or well understood.

Charles Davies:

When I was interviewing people I used an example of a terminal emulation program. Here is a program that indisputably gets events not just from the user. The normal, naïve way of writing an interactive application at that time would be to wait for a keypress, see what keypress it was, and respond to it; was it a function key, was it any other key? You'd have some horrible case statement responding to a keypress. So I would ask, 'How would you write an application where you don't know whether your next input is coming through the serial port or from the keypress?' And if they had a good answer to it they got hired, and if they didn't, they didn't.

Well we started off programming it the way that anybody would program it, you make asynchronous requests on whatever event sources you want to respond to. There are many pitfalls in doing that, for example if you don't consume that event in the right way. You end up with an event loop that's quite messy, and it's pages long, and people were making mistakes. Every event loop was buggy, and horrible bugs too, so we said 'Let's make it modular.'

Martin Tasker had the benefit of a background of programming IBM mainframes:

Martin Tasker:

I've written plenty of event-handling loops, in communications programs or command handlers where by definition you don't know what's going to happen next. Every time I wrote one of these loops I remember thinking, 'Have I got this right?' Dry running through every possibility, you used to have to tell people coming on to the team, 'No, if you handle your loop that way you're either going to double-handle some event or fail to handle some event, or you're not going to handle event number 2 if event number 2 happens while you're handling event number 1, or you're not actually going to handle event number 2 until event number 3 comes along...' These are all mistakes that everybody makes when they're writing event-handling programs. Over the lifetime of a program you tend to add in more and more events, or you remove them, and you change things around. And in those circumstances, when you're modifying existing code, it's tremendously difficult to get event-handling loops right.

Active objects were devised explicitly to solve such problems, by creating an easy-to-understand and easy-to-use mechanism for firing

off event handlers asynchronously, deliberately breaking the dependencies between events which are implied by the big, single-block switch statement which is the typical implementation. More generically, active objects enable multitasking within applications without the use of explicit multithreading.

Charles Davies:

We could have done it with threads and created a multithreaded UI, which by the way is what Java does. But the bad thing about threads is that you can pre-empt at any time, and then you've got to protect the data, because you have no idea when you're processing one thread what state the data is in. The solution was active objects, for any program that responded to events from multiple sources. So it came about because people were getting it wrong, because the old way was so complicated. So what are active objects? They're really non-pre-emptive multitasking within an application. And that is a very strong pattern. But it is also something that throws people, because it wasn't copied. It was invented here, and it's widely used, and it has been useful, but it is a particular strength of Symbian OS.

Active objects are used widely throughout the operating system, as well as providing a ready-made mechanism for developers creating native Symbian OS applications.

Martin Tasker:

Colly Myers was right, active objects are a fantastic solution. For people who know they are dealing with event-handling programs, they are an absolute joy. And the whole single-threaded nature of an application process is also great for programmers. In an event-handling system, active objects are a natural way of handling things, and they are easier for programmers to work with than pretty much all of the alternatives.

Cleanup, Leaving and Two-Phase Construction

The native Symbian OS error-recovery model evolved explicitly to handle the kinds of errors that should be expected on resource-constrained and mobile devices: low-memory situations, low-power situations, sudden loss of power, loss of connectivity or intermittent connectivity, and even the sudden loss of a file system, for example when a removable media card is physically removed from the device without unmounting. These are all likely or even daily occurrences in the mobile phone context, causing errors from which the system must recover gracefully. In contrast, for a large system these may be rare enough occurrences for system failure with an 'unrecoverable error' message to be acceptable.

The Symbian OS model is proven, playing a large part in the unrivaled robustness of the system, and going back to the earliest days of the operating system, and indeed to Psion systems before it.

Charles Davies:

We had `Enter()` and `Leave()` in the 16-bit system, which was Kernighan and Ritchie inspired. When we went to C++, the standards for exception handling were still being written, so they certainly weren't available in compilers. So we carried forward `Leave()` and `Enter()` rather than adopting native C++ exception handling, because at that time it consisted of `longjump()` and `setjump()`. It was very unstructured, and we didn't like that. We liked `Enter()` and `Leave()`, and we stuck with it.

In Symbian OS, `Leave()` is a system function (provided by the User Library) which provides error propagation within a program. Typically, `Leave()` is used to guard any calls which can fail (for conditions such as out of memory, no network coverage and disk full). The system unwinds the call stack until it finds a prior `Leave()` call wrapped by a TRAP macro, at which point the TRAP is executed and the failure is handled by the program in which it occurred.[19]

Functions which may fail because of a leave, whether because they directly invoke the action which might fail or do so indirectly by calling some other function that does, are described as 'leaving' functions. By convention, leaving functions are named with a trailing 'L', which makes it easy for programmers to see where they are invoked and trap appropriately.

The second leg of the error-handling strategy uses the 'cleanup stack' to store pointers to heap-allocated objects whose destructors will fail to be called if the normal path of program execution is derailed by a leave.[20] As well as unwinding the call stack to handle the leave, the cleanup stack is also unwound and destructors are called on any pushed objects.

The third leg of the strategy is 'two-phase construction', which guarantees that C++ construction of an object will always succeed, by moving any leaving calls out of the C++ constructor into a secondary constructor. (It is important that construction succeeds, since only then can the object's destructor be called; if the destructor cannot be called, memory may have been leaked [Stroustrup 1993, p. 311].) Again, a number of system functions are available to regularize the pattern and take care of underlying details for developers. (In its earliest implementation, two-phase construction was matched by two-phase destruction. The eventual consensus was that this was an idiom too far.)

[19] See [Stichbury 2005, p. 14] for a detailed explanation.

[20] See the discussion in [Harrison 2003, p. 150]. This is the authoritative programmers' guide.

Charles Davies:

We had an ethic that said that memory leakage was something the programmer was expected to manage. So something like the Window Server, which might be running for a year at a time, needed to make sure that if an exception was called it didn't leak memory. The cleanup stack was an invention to make it easier for people to do that. You'd have an event loop, and at the high end of the event loop you'd push things on the stack that needed to be unwound, whether they were files that needed to be closed or objects that needed to be destroyed. That was a pragmatic thing, you know. 'Let's provide something that encourages well-written applications from the point of view of memory leakage.'

Cleanup is pervasive in the system ([Harrison 2003, p. 135]), permeating every line of code a developer writes, or reads, in Symbian OS, with its highly visible trailing 'L' naming convention, its `Leave()` methods and TRAPs, and its cleanup stack push and pop calls.

For new developers, it is both highly visible and immediately unfamiliar, which leads to an immediate impression that the code is both strange and difficult. However, the conventions are not intrinsically difficult, even if the discipline may be. The purpose is equally straightforward: to manage run-time resource failures. On a small device, memory may rapidly get filled up by the user (whether by loading a massive image, downloading too many MP3s, or simply taking more pictures or video clips than the device has room for). Other resources, whether USB cable connections, infrared links, phone network signals, or removable media cards, can simply disappear without warning at any time. Mostly these hazards simply do not exist on desktop systems. On phones, they are the norm.

Martin Tasker:

I think the cleanup stack was a brilliant solution to the problem that we were faced with at the time.

Descriptors

Descriptors are the Symbian OS idiom for safe strings. ('Safe' means both type safe and memory safe and compares with C++ native C-style strings, which are neither[21]) Descriptors were invented (by Colly Myers) because there was no suitable C++ library class, or none that was readily available.

[21] Nor are Java or Microsoft Foundation Class strings for that matter, according to [Stichbury 2005, p. 55].

In principle, descriptors simply wrap character-style data and include length encoding and overrun checking. (Descriptors are not terminated by NULL; they encode their length in bytes into their header, and refuse to overrun their length.) As well as this basic behavior they also provide supporting methods for searching, matching, comparison and sorting.

Descriptors support two 'widths', that is, 8-bit or 16-bit characters, based on C++ `#define` (`typedef`) and originally designed to enable a complete system build to be switched, more or less with a single definition, between ASCII-based and Unicode-based character text support.

More interestingly, descriptors also support modifiable and unmodifiable variants and stack- and heap-based variants. The content of unmodifiable (constant) descriptors cannot be altered, although it can be replaced, whereas that of modifiable descriptors can be altered, up to the size with which the descriptor was constructed.[22]

Another important distinction is between buffer and pointer descriptor classes. Buffer descriptors actually contain data, whereas pointer descriptors point to data stored elsewhere (typically either in a buffer or a literal). A pointer descriptor, in other words, does not contain its own data. A final distinction is between stack-based and heap-based buffer descriptors. Stack-based descriptors are relatively transient and should be used for small strings because they are created directly on the stack (a typical use is to create a file name, for example. Heap-based descriptors, on the other hand, are intended to have longer duration and are likely to be shared through the run-time life of a program (see Table 3.1).[23]

Table 3.1 Descriptor classes.

	Constant	**Modifiable**
Pointer	`TPtrC`	`TPtr`
Buffer (stack-based)	`TBufC`	`TBuf`
Heap-based	`HBufC`	

See [Harrison 2003, p. 123] for a fuller explanation of the descriptor classes.

[22] Although modifiable, once allocated there is no further memory allocation for a descriptor, so its physical length cannot be extended. For example, to append new content to a descriptor requires that there is already room within the descriptor for the data to be appended.

[23] [Stitchbury 2005] contains a good overview.

Descriptors differ from simple literals, which are defined as constants using the_LIT macro, in that they are dynamic (literals are created at compile time, descriptors are not). A typical use of a pointer descriptor is to point to a literal.

Martin Tasker:

The 8-bit/16-bit aspect was ASCII versus Unicode, though, in retrospect we should have been braver about adopting Unicode straight away. But bear in mind that the ARM 3 instruction set we were then using didn't have any 16-bit instructions or, more accurately, it didn't have any instructions to manipulate 16-bit data types, so it was not efficient to use Unicode at that time. But maybe we should have had more foresight and courage, because it turned out to be a distraction. But as a kind of memory buffer, I think they were reasonably distinctive.

Given the state of the art at the time, Peter Jackson believes that the distinction between 8-bit and 16-bit was understandable but that a more naturally object-oriented approach would have been preferable.

Peter Jackson:

I think it would have been more elegant to have a descriptor that knew internally what kind of descriptor it was, whether it was the 8-bit or 16-bit variant. I never liked the fact that some of these things were done by macros.

Descriptors are not only type safe, they are memory safe, making memory overflow ('out-of-bounds' behavior) impossible. Descriptor methods will panic if an out-of-bounds attempt is detected (see Figure 3.1).

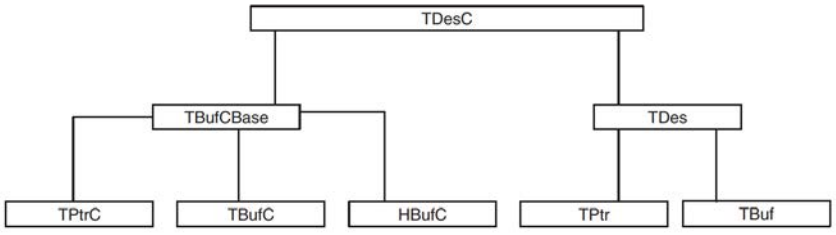


Figure 3.1 Descriptor class hierarchy

Charles Davies:

Descriptors were Colly Myers's thing, definitely, and the idea was rather like the cleanup stack, to stop people doing memory overwrites. That's a big protection against worms and other attacks, deliberate and malicious overwriting of the heap, although at the time that wasn't the driving reason to do it. We did it to stop programmers making mistakes.

C and T and Other Classes

As well as the use of the trailing 'L' (for 'leaving') and 'C' (for 'constant') to flag properties of methods, Symbian OS also uses some similarly straightforward class-naming conventions to flag fundamental properties of classes.

Martin Tasker:

If you look at the C and T types, they offer a very, very simple guide to the programmer as to how to use these types. They are as simple as Java's objects and built-ins. We don't do garbage collection because C++ doesn't do garbage collection, so we have to cope with that. We have to do it manually, but otherwise I think our conventions are as simple as Java.

The most important naming conventions are summarized as follows: [24]

- T classes are simple types which require no destructor and behave like C++ built-in types.
- C classes derive from `CBase` and should always be explicitly constructed, thus ensuring that they are always allocated on the heap. `CBase` classes also therefore require explicit destruction. `CBase` provides a basic level of additional support, including a virtual destructor, allowing `CBase`-derived objects to be deleted through the `CBase` pointer and performing cleanup stack housekeeping. `CBase` also overloads operator `new` to zero-initialize an object when it is first allocated on the heap. All member data of derived classes is therefore guaranteed to be zero on initialization.
- R classes indicate resource classes, typically a client session handle for a server session. Since an R class typically contains only a handle, it does not require either construction or destruction. R classes therefore may safely be either automatics or class members.

[24] [Stichbury 2005, Chapter 1] provides a comprehensive discussion.

- M classes are 'mixin' classes (abstract interface classes), the only form in which multiple inheritance is supported in Symbian OS.
- Descriptors are immediately recognizable as either `TPtr` pointer descriptors, or `TBuf` (stack-based) or `HBufC` (heap-based) buffer descriptors.

Manifest Constants

Symbian OS uses manifest constants – implemented as `typedefs`, that is, system-defined types – instead of the native types supported by a standard C++ compiler on standard hardware. This is partly, of course, because the cross-development model means that the eventual intended target platform is not the same as the development platform, hence the 'native' types of the platform on which the code is compiled may differ from those of the platform on which it is intended to run. The use of type definitions also has its roots in designing to support both ASCII and Unicode builds, which is now superfluous since Symbian OS has been all-Unicode since before v6.

Supporting emulator builds (that is, running Symbian OS programs on PC as well as ARM, and not just developing on PC) creates the additional complexity of requiring not one supported compiler but two (or more); originally Microsoft compilers were specified for emulator builds and GCC for ARM. More recently Metrowerks and Borland compilers have been supported and, in Symbian OS v9, ARM's RVCT replaces GCC as the 'official' ARM target compiler (although GCCE is still supported to ensure a low-cost development option). Recent initiatives such as Eclipse, for example, or the adoption of the standard ARM EABI are likely to continue to change the story of the development tools.[25] Again, using manifest constants provides the necessary level of decoupling of code from compiler dependencies.

The key classes are summarized as follows:[26]

- `TInt` and `TUint` are the generic types for signed/unsigned integer values; `TInt8`, `TInt16`, `TInt32`, and `TUint8`, `TUint16`, `TUint32` are also provided; in general, the least specific types are preferred, that is, `TInt` and `TUint`
- `TInt64` is a 64-bit integer type intended for platforms without a native 64-bit type

[25] Symbian, like Psion before it, has always assumed that mainstream development is done under Microsoft Windows, although this is not the only solution that works. There are a number of independent open-source solutions for developers wanting to work on Linux or Mac OS X.

[26] Again, [Stichbury 2005, Chapter 1] provides a comprehensive discussion.

- `TReal`, `TReal32` and `TReal64` are single- and double-precision floating-point types; again the least specific type, `TReal`, is preferred
- `TText8` and `TText16` are 8-bit and 16-bit unsigned types for characters
- `TBool` is a 32-bit unsigned Boolean type
- `TAny*` is used instead of void*.

Unique Identifiers

Unique identifiers (UIDs, implemented as signed 32-bit values) are centrally controlled in Symbian OS. One common usage of them is to identify applications and other binary and data types. UIDs, for example, are used in Symbian OS to associate data types with programs and plug-in types with frameworks. UIDs are also used as feature IDs and package IDs (for SIS files).

> **Charles Davies:**
>
> The idea was that if you had polymorphic DLLs, dynamic libraries in other words, then there are situations where the DLL is a plug-in, and it all goes very wrong if the caller doesn't get the interface it's expecting from the DLL, so we needed to characterize the interface. And we came up with the idea of using a UID to do that.

UIDs are used in a three-tier construction to build `TUidType` objects:

- UID1 – a system level identifier that distinguishes EXE from DLL types
- UID2 – a specifier for library types that distinguishes between shared library DLLs and various types of polymorphic DLL (for example FEPs and other types of plug-in)
- UID3 – the individual component ID, also used by default as the secure identifier (SID) required by platform security.[27]

UID3 is used, for example, by developers to uniquely identify their applications, and can then be used by the streams, stores and files created by that application to identify themselves. UID3 is assigned through Symbian's UID allocation database, from which third-party developers can request blocks of UIDs for use in their applications.

Platform Security introduces two new types of UID, the SID (Secure ID), which by default is identical to UID3, and VID (Vendor ID).

[27] See the discussion in [Sales 2005, p. 328].

3.8 Platform Security from Symbian OS v9

Platform Security is the system-wide security model introduced in Symbian OS v9. Providing an open, third-party programmable platform has been an important principle in the development of Symbian OS. However, openness brings with it the risk of misbehaving software (whether accidentally or deliberately misbehaving) finding its way onto users' devices. The security model is designed to protect users from that risk, while still preserving the openness of the platform.

Architecturally, Platform Security is a set of pervasive changes at all levels of the system, based on a simple conceptual model,[28] which is deliberately as lightweight as possible, and supported by the Symbian Signed certificate signing program, which provides a means for creating a formal link between an application and its origin, as well as providing a review mechanism to promote best practice in designing and writing Symbian OS applications.

Will Palmer is one of the system architects who is currently responsible for the Platform Security project.

Will Palmer:

There are three principles to Platform Security. The first principle is the unit of trust, the idea of the process being the unit of trust. Since memory is already protected per-process on the processor, that fits quite nicely, and it also has the advantage of being a 'least-privilege' approach, based on the smallest element in the operating system. The second principle is the idea of capabilities, which are in effect authorization tokens. So to be able to access a potential resource, a process needs to possess a particular capability that allows it to do so. And the third principle is data caging, which is about read and write protection of files, which protects the integrity of data as well as protecting data from prying eyes.

The essential principles are:

- processes as the unit of trust,[29] which turns trust into another process-granular system resource
- capabilities as the tokens of trust, which are required to perform actions

[28] According to [Heath 2006, p. 18], the model conforms to the eight design principles of [Saltzer and Schroeder 1975], which include economy, openness, least privilege and psychological acceptability.

[29] This is an elegant extension of the kernel's process model, in which the process is the unit of ownership of all system resources (for example, memory protection is per process).

- data caging, which protects data from prying eyes (by policing read access) or interference (by policing write access) or both.

The direct consequence of defining the process as the unit of trust is that all threads in a process share the same level of trust (which is natural, since they have access to the same resources).

The goal is to protect device users from the kinds of intentionally rogue software, or 'malware', that plague the PC world. Symbian OS for a long time avoided some of the worst threats from malware because it was typically deployed in ROM-based devices, in which the system itself cannot be corrupted (for example, it is impossible to install trapdoors or trojans in system files) because system code is stored in unwriteable ROM memory. By design, Symbian OS also protected against some of the more trivial security holes found on other systems. Descriptors, for example, make buffer overrun attacks much harder. Similarly, Symbian's microkernel architecture helps to increase security and robustness; since the trusted kernel is deliberately the smallest possible subset of system functions, there is little privileged code to exploit, and the smaller codebase is easier to review and validate.

The nature of mobile devices, especially phones, also makes them different from desktop systems. The physical access model is different (personal devices are less likely to be shared) and the network access models are different (connections are transient).

On the other hand, phones also present new opportunities for malware. If a phone, or user, can be spoofed into making a call, real money is at stake. (Premium-rate-phone-number scams are an example.) From a network perspective, the cost of network disruption is immediately commercially quantifiable in a way that Internet attacks are not.

These differences all require appropriately designed security mechanisms.

Will Palmer:

When the capability model was designed there were a set of constraints about what it had to deliver: it had to be robust; it had to be simple; and it shouldn't get in the way of the operation of a phone so, for example, you couldn't use hundreds of extra clock cycles on it, because on a small device you have performance and power constraints. Also it had to be appropriate for an open operating system: people have to be able to install additional software on their phones and it has to be simple and easy to understand.

Data caging, for example, was chosen for its simplicity and economy (in terms of clock cycles and power). Another important consideration was that mechanisms which users are quite comfortable with on desktop computers – logging on, for example – would be quite inappropriate on a phone.

Will Palmer:

Authorization based on the process–capability model is simple to understand and it fits the phone case much better than an authentication system. So in an authentication system you log on and your password authenticates you to the system, and once authenticated you can do anything permitted by your authentication level. But a phone is different: it's a single-user environment; it's in your pocket; it belongs to you. Although things are getting more complex now because of requirements coming in for administrative rights. For example, the network operator might want to change settings on the phone.

The capability mechanism is used to protect both 'system' and 'user' (i.e., application-owned) resources. Will Palmer sums up the difference neatly.

Will Palmer:

It's not that some types of capabilities are more powerful than others, they just protect different things. System capabilities protect the integrity of stakeholders and of the device, whereas user capabilities protect the user's privacy and money.

Protected APIs are tagged at method-level with the capability required to exercise them and access any underlying resources (data files, for example). The capabilities of a method are part of its interface. To use protected APIs therefore, developers must request an appropriate set of capabilities, which is done through the Symbian Signed program.

A 'signed' application is granted a set of capabilities. Application capabilities are verified by servers when protected APIs are called by applications. Unsigned software is flagged to the user at installation time as being unsigned (and therefore untrusted). Thus, while unsigned applications can assign any user capabilities to any binaries as they see fit, the user is alerted at installation time and given the option to approve the application or not. Unsigned applications cannot use system capabilities, in other words they cannot use APIs which affect the behavior of the device. Data security is provided on a per-application basis by the data-caging model.

4

Introduction to Object Orientation

4.1 Background

Symbian OS is a full-blown, from-the-ground-up, object-oriented system. In context, the decision to 'go object-oriented' was a natural step. Object-oriented ideas had been increasingly adopted in Psion's preceding operating systems, from the first Organiser products to the 16-bit SIBO operating system for the Psion Series 3. However, the decision to apply object-oriented design to the whole system, and not just to the higher user interface and application-level layers, was none the less radical for that. In particular, the decision to adopt C++ as the implementation language for the operating system was a bold one. The earlier systems (once they had evolved beyond assembler) had been written in a home-grown object-flavored dialect of C.[1] Adopting C++, which was still far from the mainstream, was, with hindsight, far-sighted though not without risk.

In 1994, when the project to create what eventually became Symbian OS started up, C++ was still a new and evolving language. C++ compiler implementations for the PC were still being pioneered by small companies such as Zortech and Watcom (the 'industrial' C++ market was still based on Unix). Microsoft had only just entered the market.[2] The language standard was still some years away. Standardized tools were even further away.[3]

The immediate consequences were twofold. First, cross-platform development was difficult (compiling on Intel for eventual ARM targets) because the low-level language bindings were not consistent across hardware

[1] See also Chapters 2 and 17.

[2] See for example the Wikipedia article ***http://en.wikipedia.org/wiki/Visual_C_Plus_Plus*** for a history of Microsoft's C++ releases. VC1.5 was the big release.

[3] Tools standardization (enabling compiler and linker interoperability across vendors, for example) depends on agreeing the low-level application binary interface (ABI). The standardized ABI for ARM processors is only now emerging into the tools mainstream.

architectures. Secondly, some language features were missing, immature, or just unsuitable for the project's purposes. While C++ was explicitly intended as a systems language, and to some extent also inherited C's low-level–high-level mantle and its long history of optimized compiler internals, some features of the language were far from optimal for small, low-memory footprint, low-power devices.[4] By and large, the language made no claim to be particularly suitable for small systems of any kind. Its roots were in big, middleware systems running on big hardware (e.g., millions of lines of code phone switches).

There were some significant consequences for the evolution of Symbian OS; many of its hallmark idioms were invented because the C++ language as it stood could not meet requirements (type-safe strings, structured exception handling, and so on) that Psion's designers considered essential for the class of device they were targeting. Subsequently, as Symbian OS has itself begun the move into the mainstream, these legacies of early language immaturity and Psion's early adoption of C++ have presented obstacles to a new generation of developers who have grown up with a standard language. Inevitably, there is pressure on Symbian OS to do better at supporting the standard language.

But it is fair to say that this problem is related to the success of Symbian OS. The pressure comes from its exposure to a much broader range of developers than in the past. It seems inconceivable, or at least unlikely, that Symbian OS would now be poised on the edge of mass-market adoption had its architects not innovated far beyond the homegrown tools and language idioms of its predecessors. The choice of C++ was a prescient one, accurately predicting what turned out to be a language juggernaut, sweeping all before it (at least until the rise of Java). There were also benefits from adopting an object-oriented methodology across the whole of the operating system.

4.2 The Big Attraction

Of all the perceived benefits of the object-oriented approach to software creation, reuse is probably the most compelling. Software is expensive. Software is unreliable. Software is complex. These are the three truisms of software development and reusability meets them all head on, or at any rate purports to.

First of all, software is expensive because it is complex. Software projects overrun because the problem at hand always turns out to be more complex than was at first thought and things prove to be harder than they looked in the plan. But if software projects can be started from a baseline of existing, already proven code or finished components, or at

[4] For example, the overhead of vtables.

least proven design, the scope for misunderstood complexity might just be reduced, and this seems to be what reuse promises. The more artifacts there are to be reused, the less the complexity, and therefore the lower the cost.

Secondly, software is expensive because it is unreliable. It is unreliable because it contains defects and it contains defects because it is complex. Reuse seems to hold promise here too because reuse improves quality by reusing proven parts. It also improves quality by reducing the complexity which causes defects in the first place.

Reuse does indeed look like the key to conquering software complexity, and this is very much how it has been sold. Object orientation claims to deliver reuse and reuse is the big attraction. In the words of [Gabriel 1996], reuse was 'the hook that grabbed the mainstream world and pulled it toward object-oriented programming'. Since effort costs money, reusing effort must save money. And since effort is error-prone, reusing effort must reduce errors.

Of course, reuse is not the privileged domain of object orientation. The earliest innovations in what were not yet called operating systems[5] were as much about code reuse as about multiplexing processors and peripherals. The same is true of the early language standardization drives, from Fortran to COBOL to C and beyond.

There are other aspects of reuse too. Reuse also occurs at project level, as every programmer quickly learns and as [Gabriel 1996] points out. Today's new problem can be understood as a variation on last week's problem, and therefore last week's solution can be adapted to become this week's solution too.

Languages, however, have the advantage of working at several levels, from the individual to the team, from program level to project level. But all languages are not equal. The clever observation that heralds the discovery of full blown object orientation is that reusing data structures counts as much as reusing algorithms. Object orientation makes this a language feature and supports it with language constructs, not just code libraries and link-time tools.

Other benefits also arise from reuse. Object-oriented analysis is a good way of modeling real-world problems. For example, object-oriented language pioneers have claimed 'real world apprehension' and 'stability of design' as two benefits which follow from the directness of the correspondence between an object model and a real-world problem domain [Madsen *et al.* 1993, p. 2]. The object approach to modeling also provides its own natural model for program organization (code is naturally granular at the object level; code can be divided between interface definitions and implementations, and so on). It probably turns

[5] Possibly the earliest example was the Supervisor program of the Manchester University Atlas computer in the late 1950s (see [Hansen 2001]).

out, too, that this way of organizing a program makes it easier to extend than more traditional organizations.

We are now some years on from object orientation's initial promise.[6] Object orientation is the industry's dominant programming methodology and software is still expensive, software projects are still delivering late (when they deliver at all: abortive projects remain at an astonishing 30% across the industry) and software is still unreliable (i.e., it cannot be guaranteed to perform its intended function without error).[7]

It would hardly be fair to blame object orientation for this, although it is tempting to ask what became of the vision of reusable components, of a black-box component industry and a free market in ready-made, reusable software parts.[8] Either the vision fizzled out or our gaze moved on. If the market ever materialized, it failed to thrive.

There are still no magical solutions [Gabriel 1996]. The truth is that simple promises rarely deliver. Reality is always more complex and more interesting than that. Object orientation, meanwhile, has enjoyed an astonishing rise and, perhaps for other reasons, remains in the ascendancy, even if the search for the 'New New Thing'[9] in reuse has moved on.

Interestingly, following what seems to be an inevitable evolutionary trajectory, the focus has shifted, or turned back, to the next level of abstraction beyond languages and beyond the meta-languages of patterns, to projects, project organization and other 'soft' or 'human' aspects of programming, with methodologies such as *extreme programming* and *agile programming* dominating the quest.

4.3 The Origins of Object Orientation

Object orientation is an approach to design and programming rather than a fixed methodology.[10] This makes it a rather loose label. At root, it is a way of thinking, a programming style, a particular approach to modeling

---

[6] It is ten years since Richard Gabriel's book was published and he was using the past tense even then.

[7] The annual CHAOS report from the Standish Group includes IT project resolution statistics. The 1994 report claimed that 31% of software projects are cancelled, with a further 16% either over budget, late or reduced in the scope of their features or functions compared to the initial specification. In 2004, the numbers were respectively 29% and 18% (see ***www.standishgroup.com***).

[8] These were the radical slogans which accompanied the announcement of the 'software crisis' and which were aimed at overturning the crisis, see [Assmann 2003, p. 6].

[9] This phrase is attributed to Netscape's Jim Clark, see [Lewis 1999].

[10] For a discussion of terminology and many interesting insights into object orientation, including the object-oriented conceptual framework, see [Madsen *et al.* 1993, p. 9]. In general, I try to follow the BETA language terminology: 'object orientation' is an outlook or perspective; 'object-oriented' is an attribute of specific tools or techniques (e.g., language implementations or analysis techniques).

the world in software. Object orientation as a programming style is distinct from any particular object-oriented language implementation.

In the first place, object orientation grew up around the need for a descriptive language for use in simulating discrete physical systems. In particular, it emerged from the work of Dahl and Nygaard at the Norwegian Computing Centre through the early and mid-1960s, which resulted in the Simula languages.[11] These ideas were in turn picked up in the early 1970s by Alan Kay's research group at Xerox PARC in California and drove the development of Smalltalk, which was initially an experiment in devising a language to teach programming concepts to children [Kamin and Samuel 1990].

Both Simula and Smalltalk (but Simula in particular) served as explicit influences for Bjarne Stroustrup, working at Bell Labs in the early 1980s and looking for a way of introducing what had become known in the literature as abstract data types into a C-style language, to try to overcome problems in writing very large systems. The specific context was large projects at AT&T, including telephone switch software (which typically were programs containing millions of lines of code). Coincidentally, an independent effort to harness the plain syntax and underlying efficiency of C to an object model was being pursued by Brad Cox and led to the appearance of Objective-C more or less simultaneously with C++.[12]

Just as both C++ and Objective-C set out with an explicit goal of creating a better C, so later twists in the story of object orientation have seen Java claiming a place as a better C++, and C# claiming in turn to be a better Java. James Gosling's group at Sun started work on what became Java in 1990, addressing the perceived shortcomings of C++ in the particular context of small, consumer devices such as set-top boxes. Java certainly achieves greater simplicity, greater language uniformity and a purer object model than C++, as well as wider goals of platform independence, language safety, and tamper resistance.

The work at Microsoft to create a better Java began in the late-1990s, as part of the Java-like managed code model for the .NET internet services framework. The result is C#, a rather small increment to Java in language terms, and a rather larger increment to C++, but one which so far is only available on the Microsoft platform. (Albeit that makes it a large marketplace.)

As well as this relatively linear evolutionary mainstream, a whole host of object-oriented languages have sprung up through several decades of research. Some have been shortlived, some have persisted, and almost all have contributed something of interest to the wider object-oriented research effort. From Beta to Sather to Eiffel to Dylan to Self to Python to Ruby, all have had some following, if only within the research community,

[11] See the discussion and timeline by Sklenar at ***http://staff.um.edu.mt/jskl1/talk.html***.
[12] Stroustrup tells the history in [Stroustrup 1994, p. 175].

and one or two have found a more permanent niche. Many other already-established non-object-oriented languages have adopted object-oriented extensions. Smalltalk style, for example, caught on in the Lisp community in the 1980s with Common Lisp Object System (CLOS), which became a model for similar extensions to languages such as Pascal, as well as more esoteric ones such as Prolog and ML. Similarly, the true inheritors of the Pascal mantle are the Modula languages, of which Modula-3 is an object-oriented language, and Oberon which again is object-oriented (and, interestingly, is not class-centric).[13]

It is hard to think of a major programming language which has not been touched, in some way or another, by object-oriented ideas.

4.4 The Key Ideas of Object Orientation

The goal of the original Simula language was to reconcile natural models of description (of complex real-world behavior) with computation (specification of algorithms which could compute such complex behavior or compute with it), to support programmed simulations. From that starting point, the key ideas of object orientation emerged.

While traditional computing languages cut the world into algorithms and data structures, object-oriented languages instead cut the world into objects, each of which encapsulates both algorithms (behavior) and data (state). Running an object-oriented program becomes more like running a physical model of the world.[14] This different approach captures a number of insights, in particular that the real world is more naturally understood as discrete and not continuous (or at any rate that we can benefit from modeling it that way) and that, in the real world, behavior comes packaged with context (context-free behavior is of formal interest only).

A few high-level principles provide the basic modeling tools of object orientation:

- *Abstraction* hides detail by finding the commonalities between things, so that difference becomes variation
- *Data hiding* hides data inside objects as state
- *Interfaces*, or behavior hiding, expresses public behavior in public protocols and hides private behavior.

Different object-oriented languages vary in the ways they support these principles, but a small number of mechanisms are almost universal (at

[13] See the official page at ***www.oberon.ethz.ch***.

[14] 'A program execution is regarded as a physical model, simulating the behavior of either a real or imaginary part of the world.' [Madsen *et al.* 1993, p. 16].

any rate in the mainstream object-oriented languages, including Smalltalk, C++ and Java):

- *Encapsulation* supports data hiding; in class-based mainstream object-oriented languages, classes are the units of encapsulation
- *Inheritance* provides the mechanism for structuring relationships within object-oriented programs and for supporting code-sharing and reuse
- *Polymorphism* (sometimes referred to as dynamic binding), the headline characteristic of object-oriented languages, is the result of abstraction and the basis for reuse; the mechanisms that enable objects to display multiple behaviors are superclass (generalization) and subclass (specialization).

While a lot of theory has evolved around object orientation, object-oriented ideas are intended to be intuitive. As Coad and Yourdon put it, quoted in [Madsen *et al.* 1993], 'Object-oriented analysis is based upon concepts that we first learned in kindergarten: objects and attributes, classes and members, wholes and parts'.

Object orientation emerged very naturally in the context of computer simulations of physical processes. The purpose of simulating a process is to understand it; but, in order to simulate it, it must be modeled and modeling requires understanding. To break the regression, think of modeling as a way of transforming one kind of understanding into another kind (information in this respect is like energy or matter: it resists lossless compression). A model reduces a problem in a systematic way to recognizable objects, parts, and the relationships between them, allowing a deeper understanding to emerge from the complex dynamics which arise in the running system from the interactions between objects. A good model represents an object in a way which reveals more information about the object than was available without the model.

Arguably, all programming is based on the principle of abstraction[15] (all problem decomposition is abstraction by one means or another), but every language lends itself to a particular programming style (the one it makes easiest). Each language provides a different conceptual toolkit and encourages and enables different design and implementation techniques. Abstraction, inheritance and polymorphism are the essential characteristics of object-oriented languages.

'Abstraction', as [Koenig and Moo 1997, p. 9] rather neatly puts it, 'is selective ignorance'. Inheritance and polymorphism are what make abstraction in object-oriented languages different from abstraction in other programming languages.

Inheritance builds on the 'is-a' relationship as a way of capturing similarities between types. Objects in a program are defined into a

[15] There is an interesting discussion of abstraction in [Koenig and Moo 1997, p. 75].