Lappeenranta University of Technology
Department of Information Technology
Master's Thesis

Application Size Optimisation in Mobile Environment

The topic of the Master's Thesis has been accepted on November 14th, 2001 by the Department Council Meeting of the Department of Information Technology.

Supervisor:
Prof. Jari Porras

Instructor:
M.Sc. Kimmo Hoikka

Lappeenranta, January 21, 2002

Heikki Pora
Punkkerikatu 6 as 24
FIN-53850 Lappeenranta
+358 40 737 0726
heikki.pora@digia.com

ABSTRACT

Lappeenranta University of Technology
Department of Information Technology
Heikki Pora
Application Size Optimisation in Mobile Environment
Master's Thesis, 2002
71 pages, 19 figures, 10 tables
Supervisor: Professor Jari Porras
Keywords: Optimisation, Refactoring, Design patterns, Software process, Compression, Symbian, C++

The thesis studies methods, practices and patterns that lead to smaller software. The thesis explores concrete means for software size optimisation on the Symbian Platform. The focus is on C++ software that is designed to run on mobile phones and other wireless devices.

During the thesis a real, mobile, end-user application was analysed and optimised in size. The used optimisation methods are discussed and the results analysed. Based on the case study, recommendations for small memory software development are presented.

The importance of a good technical architecture was noted. C++ class derivation was surprisingly discovered as the biggest source of binary bloat in Symbian OS applications.

It requires skill and discipline to create small software but at least it can be done. The attitude of developers is the greatest obstacle. Many people just do not care about the size of the software they write.

TIIVISTELMÄ

Lappeenrannan teknillinen korkeakoulu
Tietotekniikan osasto
Heikki Pora
Application Size Optimisation in Mobile Environment
Diplomityö, 2002
71 sivua, 19 kuvaa, 10 taulukkoa
Tarkastaja: Professori Jari Porras
Hakusanat: Optimointi, refaktorointi, suunnittelumallit, ohjelmistoprosessi, pakkaaminen, Symbian, C++
Keywords: Optimisation, Refactoring, Design patterns, Software process, Compression, Symbian, C++

Työssä tutkitaan menetelmiä, käytäntöjä ja oliosuunnittelumalleja jotka johtavat ohjelmistojen koon pienentymiseen. Työssä tutkitaan konkreettisia keinoja ohjelmistojen koon optimointiin Symbian-alustalla. Työ keskittyy C++ ohjelmistoihin jotka on suunniteltu toimimaan matkapuhelimissa ja muissa langattomissa laitteissa.

Työssä esitellään, analysoidaan ja optimoidaan todellinen, loppukäyttäjille suunnattu, langaton sovellus. Käytetyt optimointimenetelmät sekä saadut tulokset esitellään ja analysoidaan. Esimerkkisovelluksen toteuttamisesta kertyvien kokemusten perusteella esitetään suosituksia langattomaan sovelluskehitykseen.

Hyvän teknisen arkkitehtuurisuunnitelman todettiin olevan merkittävässä roolissa. C++ -kielen luokkaperinnän huomattiin yllättäen olevan suurin ohjelmatiedostojen kokoon vaikuttava tekijä Symbian–käyttöjärjestelmässä.

Pienten ohjelmien tuottamisessa vaaditaan taitoa ja kurinalaisuutta. Ohjelmisto-kehittäjien asenteet ovat yleensä suurin este sille. Monet ihmiset eivät vain välitä kirjoittamiensa ohjelmistojen koosta.

PREFACE

This thesis was written for Digia Inc. in Lappeenranta branch office. I became interested in formal application size optimisation in late summer 2001. I also had previous experience in the subject from my history with software development. The writing was mostly done during late autumn 2001 and the case study implementation started at the beginning of December 2001. At the end, I was surprised how easily this thesis was created.

I would like to thank the following people for helping me during the writing process. Kimmo Hoikka, the instructor of this thesis, for rigorous inspection and good guidance. Jari Porras, my supervisor, for support and comments. Katja, my wife, for inspiring and motivating me. Aurora, my newborn daughter, for being the ultimate reason to get the work done in time.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

ABI	Application Binary Interface
API	Application Programming Interface
ASIC	Application-specific Integrated Circuit
DBMS	Database Management System
DCT	Discrete Cosine Transform
DLL	Dynamic Link Library
DMA	Direct Memory Access
DSP	Digia Software Process
EE	Enterprise Edition
FEP	Front End Processor
GIF	Graphics Interchange Format
GSM	Global System for Mobile communication
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IIS	Internet Information Server
IPC	Inter-Process Communication
ISDN	Integrated Services Digital Network
JPEG	Joint Photographic Experts Group
KB	Kilobyte (= 1024 bytes)
LCD	Liquid Crystal Display
LZ	Lempel Ziv
LZW	Lempel Ziv Welch
MB	Megabyte (= 1024 Kilobytes)
MBM	Multi-Bitmap
MMC	MultiMediaCard
MMU	Memory Management Unit
OS	Operating System
PC	Personal Computer
PCM	Pulse Code Modulation
RAM	Random Access Memory

RAS	Remote Access Service
RISC	Reduced Instruction Set Computer
RLE	Run Length Encoding
ROM	Read Only Memory
SDK	Software Development Kit
SMS	Short Message Service
SQL	Structured Query Language
TCP/IP	Transmission Control Protocol / Internet Protocol
UI	User Interface
UML	Unified Modeling Language
VFTABLE	Virtual Function Table
WAP	Wireless Application Protocol
WAV	Wave
WML	Wireless Markup Language
WWW	World Wide Web

1 INTRODUCTION

1.1 Background

Growing software size has been a continuous trend in the software industry for the last two decades. Every time a more powerful computer is made, software vendors update their software with new features to "take advantage of the new possibilities". The same effect can now be seen on the embedded software market. Applications are becoming too large in terms of code and binary size.

The mobile phone industry is moving towards platforms that are open to third party development. Nokia, Sony-Ericsson and Motorola among the others have started to utilise, or at least have licensed, an open platform for their future products. Third party developers and subcontractors mean that the vendors have less control over the quality, size and performance aspects of the pieces of software installed into their new mobile phones.

It is harder and harder to get qualified professional staff as the industry has already digested most of the available workforce. Not all developers have the skills to put up against the modern, complex, mobile platforms. The lack of experience shows quickly on the size, maintainability and performance aspects of written software. Many produced applications will need heavy optimisation in order to be usable.

1.2 Objectives and restrictions

The objective of this thesis was to study methods for shrinking the size of applications running on the Symbian Platform /1/. In addition, the effect of optimisation on the software process as a whole had to be considered. Size optimisation nearly always affects software performance but it is not the primary concern here. The work is based on the Symbian Platform but most of the material is not strictly Symbian specific and applies to any modern small memory environment /2/. The Symbian OS version in question is release 6.1.

1.3 Structure of work

The thesis starts with an overview of the Symbian Platform in Chapter 2. Chapter 3 describes the problems and restrictions encountered in software development on a mobile environment. Concrete methods for application size optimisation are given in Chapter 4. An optimisation process is developed in Chapter 5. The methods are also fitted into a standard software development process. Chapter 6 highlights the key points in optimising a real application with the said methods. The work done in the thesis is discussed and concluded in Chapter 7.

2 THE SYMBIAN PLATFORM

This chapter describes the Symbian Platform environment, its structure and what hardware it supports. In addition, the application development facilities are introduced. Detailed information about the platform and its licensing is available at the Symbian website /1/.

2.1 The Symbian OS

The Symbian Operating System (*OS*) is a modern 32-bit operating system designed for low-cost, low-power mobile devices. Its history goes back to the early 1980's and to 8-bit hand-held organisers manufactured by Psion /3/. The operating system was long known by the name EPOC and was just lately changed to Symbian OS. The latest publicly available version is 6.1. At present, Symbian OS based mobile phones are available from Ericsson and Nokia. Figure 1 shows the top-level components of the OS and the division to reference designs. The components are explained subsequently in 2.1.1 and 2.1.2.

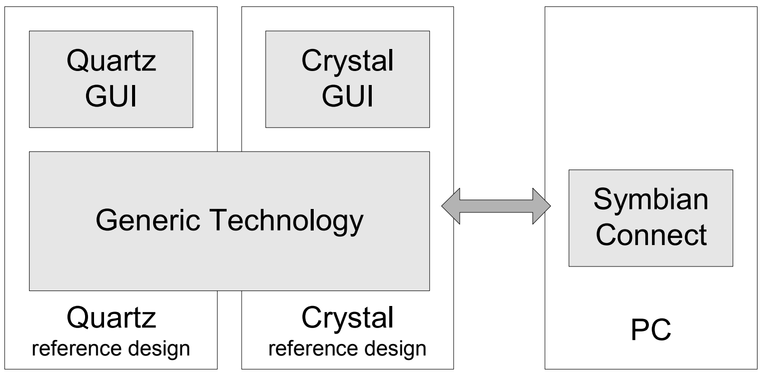


Figure 1. Symbian OS top-level components.

2.1.1 Reference designs

A reference design defines the form-factor, Graphical User Interface *(GUI)* outlook and input methodology of a device. The Symbian OS is currently delivered as two reference designs, Quartz and Crystal. Companies who license the operating system can also implement their own “reference designs” if they desire. Quartz and Crystal are just the ones available ready-made from Symbian.

Quartz. Quartz is a tablet communicator with pocket-sized, palmtop form factor. It has a 240x320 sized colour touch screen with built-in handwriting recognition. All operations are carried out with a pen, as there is no keyboard. An integrated, task-based, application suite is also provided ready. Quartz is very similar to the popular Palm handheld devices. Figure 2 shows an example of the Quartz user interface. The original picture is from the Symbian website /1/.

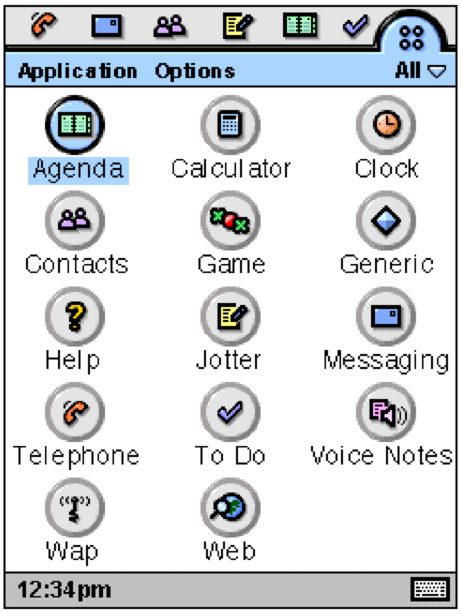


Figure 2. Quartz Application Launcher.

Crystal. Crystal is targeted at professional and power users. It has a full keyboard and a 640x200 sized colour screen, with programmable soft keys. Crystal devices may or may not support pen-based operation. Crystal is delivered with a full set of common office applications. Figure 3 shows an example of the Crystal user interface. The picture is a screen capture from the Nokia 9210 emulator.

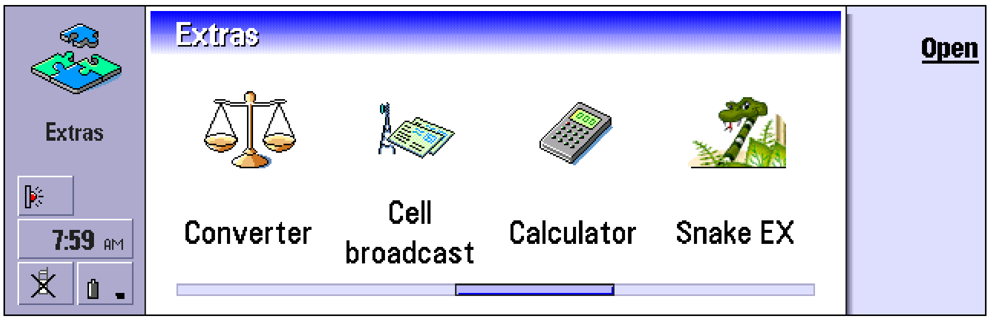


Figure 3. Crystal Extras Application Launcher.

2.1.2 Generic Technology architecture

Symbian's Generic Technology is a large collection of Application Programming Interfaces *(API)* and services that are shared between all reference designs. It forms the core of the Symbian OS by providing layers of high-level functionality on top of the OS kernel.

Figure 4 shows the major components of Symbian's Generic Technology both on the Symbian OS device side and on the Personal Computer *(PC)* side.

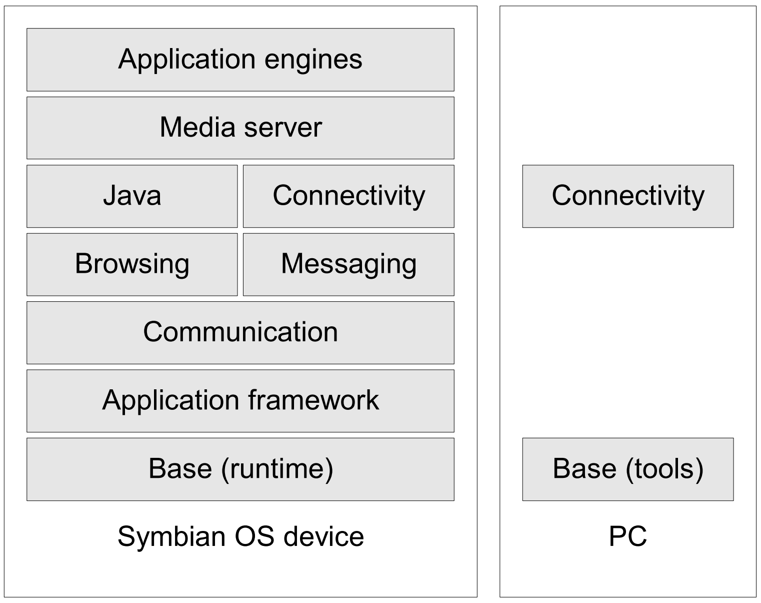


Figure 4. Generic Technology components.

Base. Base provides the fundamental runtime system, the tools required to build it, and lowest-level security. The operating system kernel, file systems and user library are the major sub-components of the base. The Symbian OS kernel employs a microkernel architecture where a minimal functional core is separated from extended functionalities and customer-specific parts.

Application framework. Application framework consists of middleware APIs for data management, text handling, clipboard, resource file support, internationalisation, and core GUI components. Application Architecture, Window server, View server and the GUI component framework are the major sub-components.

Communication. Three low-level servers are contained in the communications infrastructure: Comms server, Socket server and Telephony server. Support for the most common communication technologies and protocols is provided based on the mentioned servers. Wide-area communications stacks include Transmission Control Protocol / Internet Protocol *(TCP/IP)*, Global System for Mobile communication *(GSM)* and Wireless Application Protocol *(WAP)*. Serial communications, infrared and Bluetooth represent industry standard personal-area communication technologies.

Browsing. Browsing includes complete browsers for both Wireless Markup Language *(WML)* used in WAP services and Hypertext Markup Language *(HTML)* used in World Wide Web *(WWW)* services. The HTML browser can be extended by writing plug-in modules just like traditional web browsers in the PC world.

Messaging. Messaging is based on the messaging server. Support for new message types and transports can be added as Message Type Module (*MTM*) plug-in modules. A universal inbox and folder management are provided. Outbox handling includes repeated attempts to send messages, if necessary. Internet e-mail, GSM Short Message Service *(SMS)* and fax are supported by default.

Java. Full Java implementation is provided within the Java 2 Micro Edition framework. This includes PersonalJava 3.0 specification Java Virtual Machine based Java runtime system, with JavaPhone 1.0 APIs /4/.

Connectivity. Connectivity offers a communications framework for connecting with a PC running Symbian Connect software. Converters and viewers for foreign data formats, like Microsoft Word e-mail attachments, are located on both ends of the link.

Media server. Media server makes available a rich set of multimedia capabilities such as audio recording and playback, and image related functionality. Popular media formats such as Wave *(WAV)*, Joint Photographic Experts Group image file *(JPEG)* and Graphics Interchange Format *(GIF)* are supported by default. New media types can be implemented and added to the Media server as plug-in modules.

Application engines. Application engines layer provides core functionality for all standard Symbian applications. Each reference design provides its own user interface code on top of the standard engines. Third-party application developers can also utilise the engines' services in their own applications. The engine collection includes contact management, a spell-checker and a schedule manager, to name a few.

2.2 Supported hardware platforms

Symbian OS is designed to run on ARM equipped platforms, though porting to other systems as well should not be too hard. ARM is a widely used embedded Reduced Instruction Set Computer *(RISC)* processor and microcontroller family. It is manufactured and designed by ARM Ltd. /5/.

2.2.1 ARM processor family

The ARM family incorporates a large number of processor cores, of which a few are suitable for Symbian OS use. ARM720T and ARM9E are good examples of typical processors used in Symbian OS based devices. They both contain, in addition to the cached core, a memory management unit and a flexible bus for controlling peripheral devices. The cores are optimised for cost and power-sensitive applications. Figure 5 shows the simplified structure of a hardware implementation with an ARM720T /5/. The ARM core, buses and the Direct Memory Access (*DMA*) controller are contained in a single Application-specific Integrated Circuit (*ASIC*) microchip.

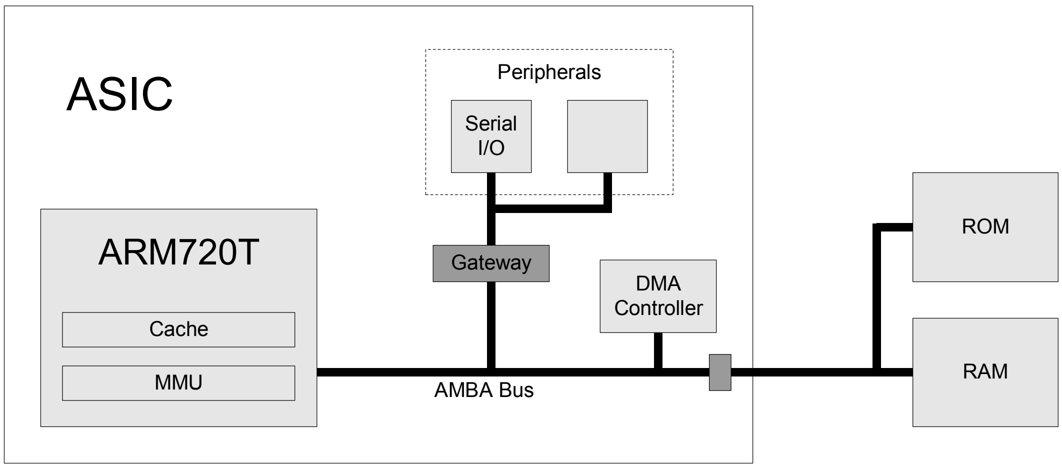


Figure 5. Typical implementation with an ARM720T.

ARM7 and ARM9 generations support ARM4 and ARM4T instruction sets. The ARM4 instruction set is a typical 32-bit RISC instruction set with additional support for 16-bit quantities and sign extension on 8-bit loads. ARM4T adds support for a so-called Thumb mode. The Thumb architecture compresses many instructions to 16-bit opcodes, achieving a much higher code density. There are fewer operations available in the Thumb instruction set than in ARM4. Therefore, more Thumb instructions are needed for the same task /6/. More Thumb characteristics are described in Chapter 4.2.1.

2.2.2 Binary formats

In practice, the ARM instruction set is insufficient to specify how a compiler should behave. An Application Binary Interface *(ABI)* defines the general rules for calling functions and the like. As one would expect, there are ABIs for ARM and Thumb code. There is also an interworking ABI that compiles to ARM code but can call, or be called by, Thumb code.

The application developer can choose the format of the application from the following three ABIs

- *ARM ABI* - requires ARM or ARMI ROM
- *Thumb ABI* - requires Thumb or ARMI ROM
- *ARMI ABI* - will run on any ROM

Figure 6 shows how each ABI relates to the different ARM instruction sets. The BX instruction is used to switch the mode. The ABIs and their usage are discussed in more detail on the Symbian website /1/.

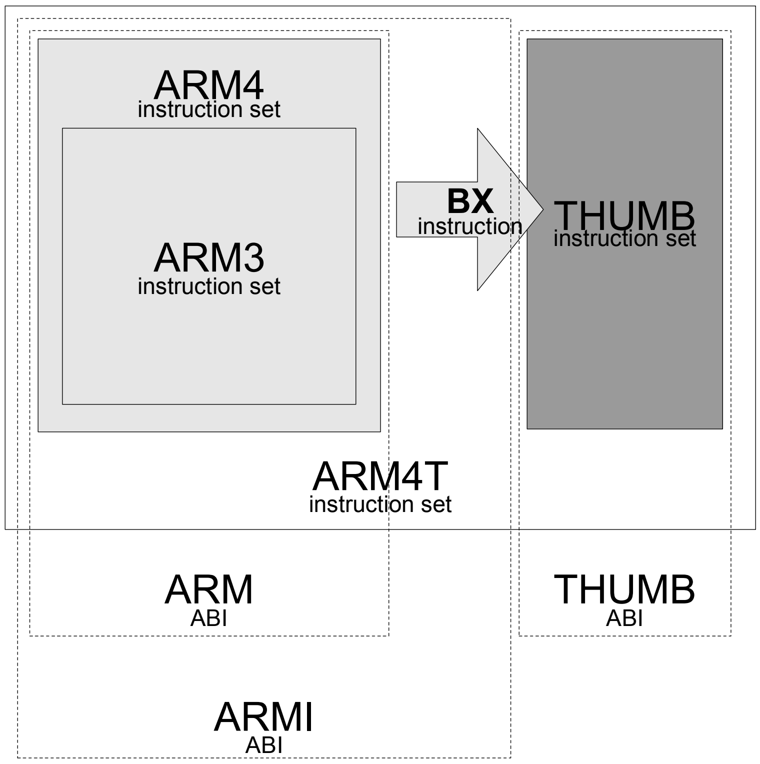


Figure 6. ARM instruction sets and ABIs.

2.3 Application development

Software for the Symbian Platform can be developed by using either C++ or Java. This thesis concentrates solely on applications written in C++. Detailed Java development information is available on the Symbian website /1/.

Software is developed on Microsoft Windows based PC workstations and then cross-compiled to a form suitable for the mobile device hardware. The Symbian Platform Software Development Kits *(SDK)* contain all the tools needed for small-scale mobile application development.

2.3.1 Toolchain

The Symbian C++ SDK toolchain is based on various small tools that are realised mostly as Perl scripts. These are used for building makefiles and for controlling all operations required by the build process. ActivePerl by ActiveState is included for the purpose. A tool for converting makefile specifications into makefiles or into Microsoft Visual C++ workspaces is also provided.

Microsoft's Visual C++ compiler is used to build and debug software for the Symbian OS emulator introduced in 2.3.2. Visual C++ is naturally not included in the distribution. The free GNU C++ compiler, GCC, is used to cross-compile software for ARM-based target machines. GCC has been customized for Symbian OS requirements by Cygnus, Ltd. The source code for this custom implementation is available from Symbian free of charge.

In addition, there are several compilers and converters for resource files, bitmaps, help files and other program resources. These are needed because the Symbian OS uses proprietary file formats for most of the data files. Compressed installation packages can be generated by the installation packager tool. The Symbian installation system works seamlessly in all Symbian devices regardless of the particular OS version and DFRD.

In order to lower the rather steep learning curve of Symbian OS software development, a handy application wizard for Visual C++ is also provided. Application information files and icons can be created using Java based graphical tools.

2.3.2 The Symbian OS emulator

The emulator is an implementation of the Symbian OS kernel and device drivers. It uses Microsoft Windows and PC hardware to emulate a real Symbian OS device. The emulator does not emulate a full ARM hardware implementation. Instead, it runs Intel x86 machine code and therefore does not provide support for native ARM binaries. The performance of applications running on the emulator is dependent on the host PC and does not reflect any target hardware performance characteristics.

The emulator uses a single window to display the device's screen, surrounding plastics, keypad, and LED indicators. Figure 7 shows the Quartz emulator outlook. PC keyboard and mouse can be used to provide key input and pointer input respectively. A directory tree in the PC's file system is used to emulate the Symbian OS file system. Windows Remote Access Service (*RAS*) provides access to modem and Internet connections.

Symbian OS C++ programs are built for the emulator from the same C++ source code as is used for real devices. The C++ source code is built as Windows Dynamic Link Libraries (*DLL*) containing Intel x86 machine code. Applications are then rebuilt in native machine code, using the ARM instruction set, to run on a real ARM-based device.

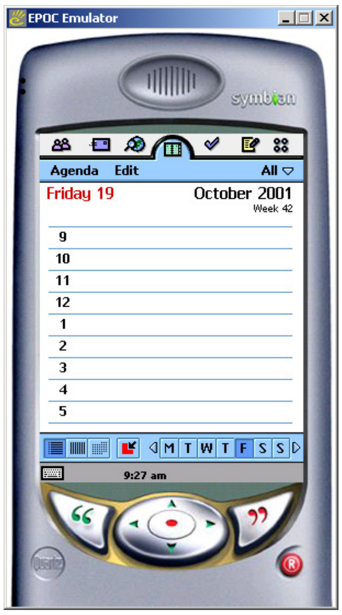


Figure 7. Symbian Quartz SDK emulator.

3 PROBLEM DOMAIN OVERVIEW

This chapter describes what the current mobile applications are like, what are their demands, and what kind of restrictions they face. A mobile phone is a very unforgiving environment and it poses many challenges and potential pitfalls for the developer.

3.1 Mobile applications today

The current mobile applications are best introduced by having an example case. This example is a real-world Symbian OS application that represents an image catalogue. Images can be listed as thumbnail images, panned, rotated, and viewed at different zoom levels. The images can also be moved around the file system and be sent via e.g. infrared link to another device.

This application is a model example of an application that has grown over time a little bit too much in size. It seems to have quite a lot of unnecessary “code bloat”. Table 1 lists some of its key characteristics from the software developer’s perspective.

Lines of code	27 700
Number of C++ classes	32
Lines of code / C++ class	133 to 2 501 (avg. 866)
Number of GUI views	2
Number of components (DLLs)	4
Run-time footprint, see 3.4.1	140 to 416 Kb (avg. 280)
On-disk footprint, see 3.4.2	77,5 Kb

Table 1. Properties of the example application.

It can be seen that the classes of the application are quite large. The largest of the classes is even possessing a total of 2501 lines of code. The classes are probably trying to do too much and might have a lot of duplicate code /7/. Otherwise, the application does not seem to be really bad. The on-disk footprint is below 100 kilobytes and the memory consumption is on average below 300 kilobytes.

Figure 8 shows the run-time memory consumption curve while the application is being used for some basic tasks. The first GUI view (VIEW 1), which displays a list of available images, consumes a steady 140 kilobytes. The second GUI view (VIEW 2), which is used to display and manipulate a single image, consumes up to 416 kilobytes when it is responding to user commands. There is somewhat too much variation on the curve and the application would be better off with a less aggressive memory strategy. Large variations potentially cause exceptions that, in turn, need more code to handle them. On the other hand, allocating all memory at the beginning and consuming lots of memory all the time is not good either.

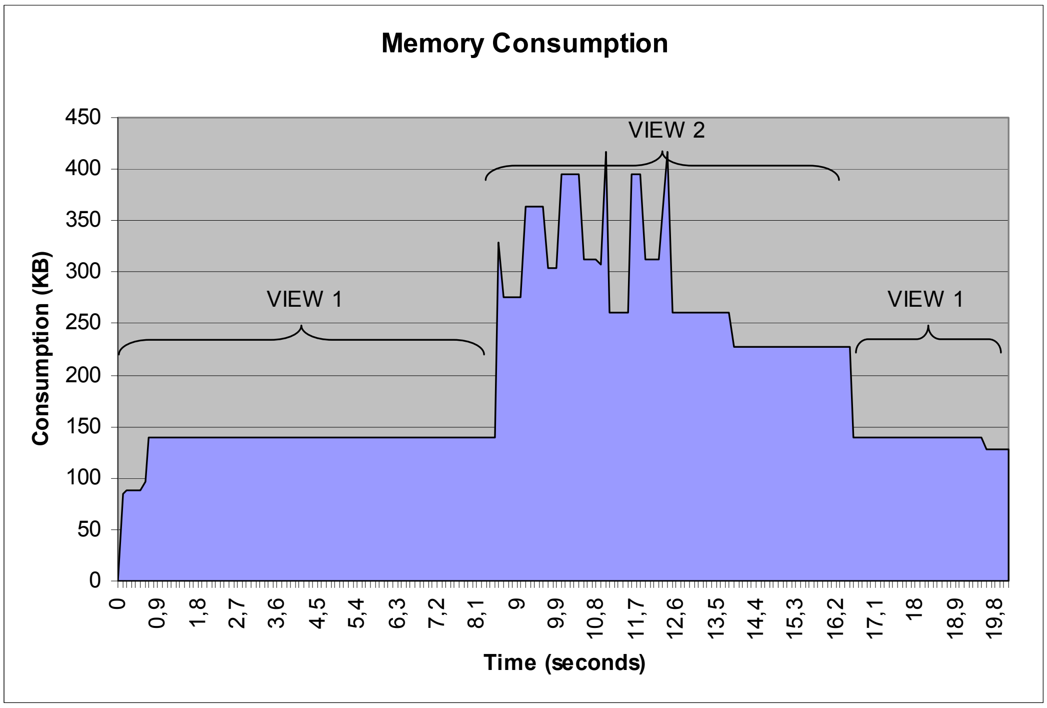


Figure 8. Memory consumption of the example application.

3.2 Demands

Application and system developers face many demands from the users and from the device manufacturers. Both are interested in the product’s value-for-money.

3.2.1 User experience

User experience, or usability, is one of the key factors when people are making a purchase decision over a new mobile phone model. Potential buyers will quickly fade away if the user interface is awkward, illogical and sticky slow. Performance may even be as important as the feature set. On the other hand, many people will only purchase a new phone if their old model is clumsy.

Common operations such as navigating the user interface must be as swift as possible. The number of long running operations should be minimised and be done as background processing without interrupting the user. A delay of twenty milliseconds is indistinguishable between key presses but if the delay is close to half second it quickly becomes irritating.

Applications must handle all kinds of special situations. The device may run out of memory. The battery might die. The network connection may suddenly go away. All these events must be handled gracefully or the user gets irritated, again.

3.2.2 Benchmarking

Device manufacturers, such as Nokia and Ericsson, benchmark their products with the following methodology. When a new competitor device enters the market, it is examined very carefully. First, the features of the device are counted based on its user manual. Then the device is broken to bits and pieces and the amount of installed memory is checked. The resulting benchmark value is finally calculated by dividing the number of features by the amount of memory. The higher the benchmark value is the better the device is considered to be. Therefore, it is important to get the most features in the smallest space. The process is illustrated in Figure 9.

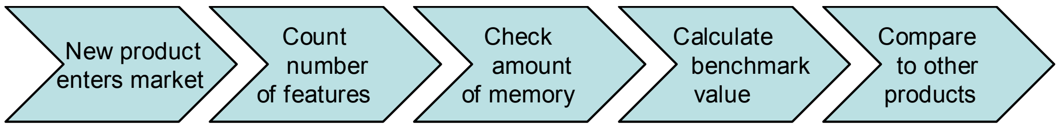

New product enters market
Count number of features
Check amount of memory
Calculate benchmark value
Compare to other products

Figure 9. The benchmarking process.

3.3 Restrictions

Mobile phones are very restricted in terms of processing power, disk space or execution memory capacity. Application developers must be aware of the limitations if they want to implement software that behaves properly in the environment.

3.3.1 Battery lifetime

The devices operate on battery power. For example, on the Nokia 9210 communicator the battery lifetime ranges from just 4 hours to a total of 230 hours /8/. Changing electric current requirements of different components causes the large difference. Three of the most power consuming components are the GSM circuitry, the Liquid Crystal Display (*LCD*) and the ARM processor. The components try to minimise their power consumption internally and, for example, the GSM circuitry and the LCD might even be turned off completely.

GSM circuitry. Being a mobile phone, the device needs to communicate with the GSM network base stations. The amount of energy consumed varies depending on the location of the device and the application programmer cannot really affect it. In general, the farther the device is from the base station the more power it must use for transmission.

Display. An LCD consumes a lot of energy and therefore must be dimmed or even switched off after a short period of user inactivity. The more shades or colours a display has to show the more energy it consumes. For example, a grey scale display mode with 16 shades consumes about twice the energy needed for a 4-shade display mode. Even the displayed pixel pattern affects the consumption. The programmer must use as low specification display mode as possible. However, changing the display mode is not necessarily possible as is the case with the Nokia 9210.

ARM processor. The ARM processor family is designed to consume little power. It is able to slow down or even go to an idle state in order to conserve energy if it is not used for a period. However, this is not possible if the processor is in full use all the time. The applications must be implemented in such a way that the processor can go to a power-save mode. The Symbian OS has many power management functions that automatically control the processor power-save features but it is still crucial that the programmer does not implement processes that run forever.

For example, the ARM7100 consumes 72 mW in full operational mode. In idle mode, with the CPU stopped but other systems running, it consumes 33 mW. In standby mode, with only a 32 kHz clock running, it consumes just 33 µW /6/.

In addition, the number of memory modules in use, and the frequency and type of access to the memory modules, may affect power consumption even significantly. It is impossible to estimate the power requirements by analysing the software, but as a rule of thumb, the less memory usage the better.

3.3.2 Memory capacity

There are usually two or three kinds of memory installed into modern mobile phones. First, there is Read Only Memory (*ROM*) or Flash memory for storing the operating system and its components. Second, there is Random Access Memory (*RAM*) for the execution of applications and storage of user data. Third, there may be an exchangeable expansion memory card for the user's own files. The MultiMediaCard (*MMC*) /9/ and the Memory Stick /10/ are common expansion card standards.

For example, the Nokia 9210 has 16 Megabytes (*MB*) of Flash memory, 8MB of RAM and a 16 MB MMC. 2 MB of the Flash memory serve as a disk drive for user data. After the device is booted, it has approximately 3 MB of free execution memory left.

This means that two separate applications that each take over 1.5 MB cannot be running at the same time.

Because the devices are designed to be always on, it is very important for the application not to “leak memory”. Leaking memory means that an application allocates more memory than it frees during its lifetime. This leads slowly to a point when there is no free memory at all. The Symbian OS goes to great lengths to protect the system and applications from this kind of behaviour. Yet, the programmer is responsible for making sure that his/her application behaves well. See 4.3.1 for a detailed description of the mechanisms.

Memory capacity has also an effect on battery lifetime described in 3.3.1. Installing more memory modules means larger energy consumption and shorter battery lifetime. Increase in the amount of memory means an increase in the manufacturing costs also. Albeit not substantial on one device, the increase becomes significant when several million devices are being produced.

It is also a fact that there is only a limited number of memory modules manufactured each year. Nokia’s representatives have publicly estimated that the demand for Flash memory will exceed production in just a few years time.

3.4 Application footprint

Application footprint can be seen as two distinct areas, *run-time* and *on-disk* footprint. Run-time footprint is limited by the amount of available execution memory (RAM) and on-disk footprint by permanent storage (usually ROM or MMC).

3.4.1 Run-time footprint

Run-time footprint means the execution memory consumption, or dynamic size, of the application at any one time. Run-time footprint usually varies more or less during the execution. In the Symbian OS environment the run-time footprint can be seen as the sum of the areas shown in Figure 10. The figure also shows the relative proportions of the said areas contained in a small application.

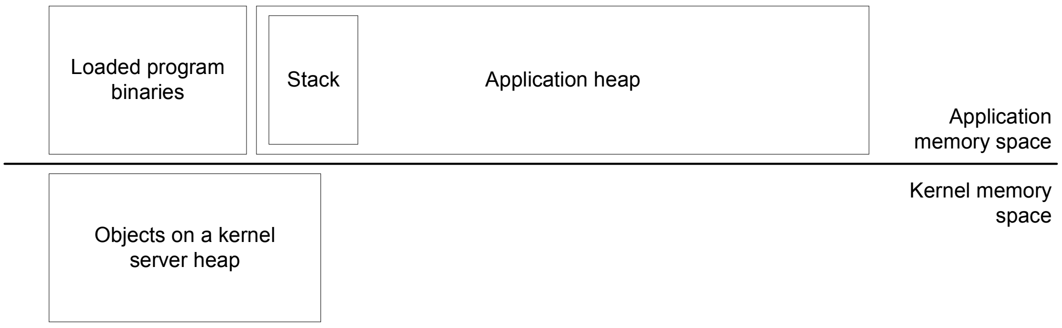


Figure 10. Application run-time footprint.

Figure 11 shows how the different parts are located on the application process' memory chunks, and how the chunks are located on the Symbian OS memory map /11/. The memory map does not represent the physical addresses but the *linear* address space that the application process sees. When a process is in execution its writable data chunks are remapped from the *home section* (at the top) to the *run section* (at the bottom). Kernel side objects (i.e. data), user code chunks, and user data chunks with static data, always reside in the home section. An exception to this is user code located in ROM. Such code does not have a code chunk at all and the code is executed straight out of ROM.

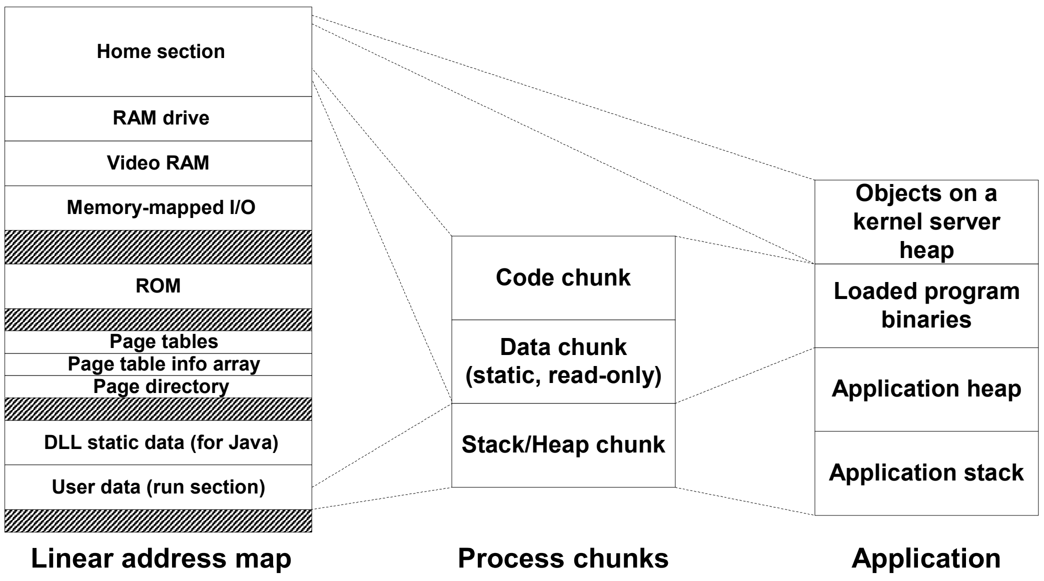


Figure 11. Run-time process chunks and the memory map.

Application heap. Heap is special structure that keeps track of the allocated memory. It can be though as a mini file system for RAM. The virtual size of a heap is four gigabytes. Memory is allocated from the heap as and when required. It is used for objects that are created or manipulated at run-time, and which cannot be stored on the stack because they are too big, or because their lifetimes do not coincide with the function that created them.

Application stack. Stack is a memory block automatically allocated for each application from the same memory chunk as heap. Stack is usually eight kilobytes in size but can be changed at compile time. It is suitable for storing small fixed-size objects whose lifetimes coincide with the function that creates them. All automatic variables in C++ are created on the stack. Applications that use stack-based variables extensively risk overflowing the stack and thus creating an exception. Stack grows downwards as opposed to heap which grows upwards. This is to prevent the stack from overwriting heap memory in an overflow situation.

Loaded program binaries. Program binaries are constant and do not change. They are automatically loaded when needed by the operating system and are usually disposed afterwards from the execution memory.

Objects on a kernel server heap. Each kernel server process in Symbian OS has its own heap. When an application uses the server's services it simultaneously causes memory to be allocated from the server's heap as opposed to the application's own heap. For example, if an application creates a bitmap image object it must be stored somewhere. The Font and Bitmap server takes care of this. The objects do not consume the application heap, but rather the server heap. This eats up the total system memory in any case.

3.4.2 On-disk footprint

On-disk footprint is the total size of all the files comprising the application counting any related user data files. It is also called the *static size* of the application. On-disk footprint usually grows slowly over time as the user creates new files with the application.

- Program binaries (.app, .dll, .exe)
- Resource and application information files (.rsc, .aif)
- Data files, images, multimedia

Program binaries. Program binaries form the body of the application. They are the result of the C++ source code compilation and linking process. Binaries grow when new features are added to the application.

Resource and application information files. Resource files contain the GUI element description structures of the application. In addition, strings, numeric parameters, file paths and other parametric items are stored there. On a localised device, there is usually one instance of a resource file for each supported language. Resource files must therefore be carefully balanced by usefulness and size. An increase of 100 bytes in a system where there is support for 10 languages makes the enlargement a total of 1000 bytes. Application title, icons and capability information are stored in the application information files.

Data files, images, multimedia. On productivity applications such as word processors, spreadsheets and contact managers, the number of files belonging to the application itself is usually rather small. The amount of data grows over time due to the user creating new data files, e.g. text documents. On the other hand, media intensive applications such as games incorporate a much larger amount of data by default. Nevertheless, when they are used the number of data files rarely increases, as there are no or very little data to save.

3.5 Code bloat

The term "code bloat" is often used to describe popular Microsoft Windows software. This refers to the fact that Windows programs increase in size, sometimes substantially, with each new release. This is why it is common to replace the PC hardware when upgrading to the next major release of Windows. The machine's processor has more instructions to execute to accomplish the same task and the hard disk must store more code. Table 2 shows how Windows has grown over time, measured in lines of code /12/.

Microsoft Windows version	Lines of code
Core NT v3.51	5 million
NT v3.51 + IIS	8 million
Core NT v4.0	8 million
NT v4.0 + IIS	12 million
NT v4.0 Enterprise Edition	16 million
2000 Professional Beta 1	27 million
2000 Professional Beta 2	30+ million
2000 Professional	35 to 40 million
Windows XP	45 million

Table 2. Size of Microsoft Windows in lines of code.

Code bloat is not only the curse of Windows software; it more or less affects all software. In an environment where hardware upgrades are not possible this is not acceptable behaviour.

There are many reasons that lead to bloated code. Figure 12 shows a dozen most common problem areas by grouping them into three categories: software lifecycle, environment and the human factor.

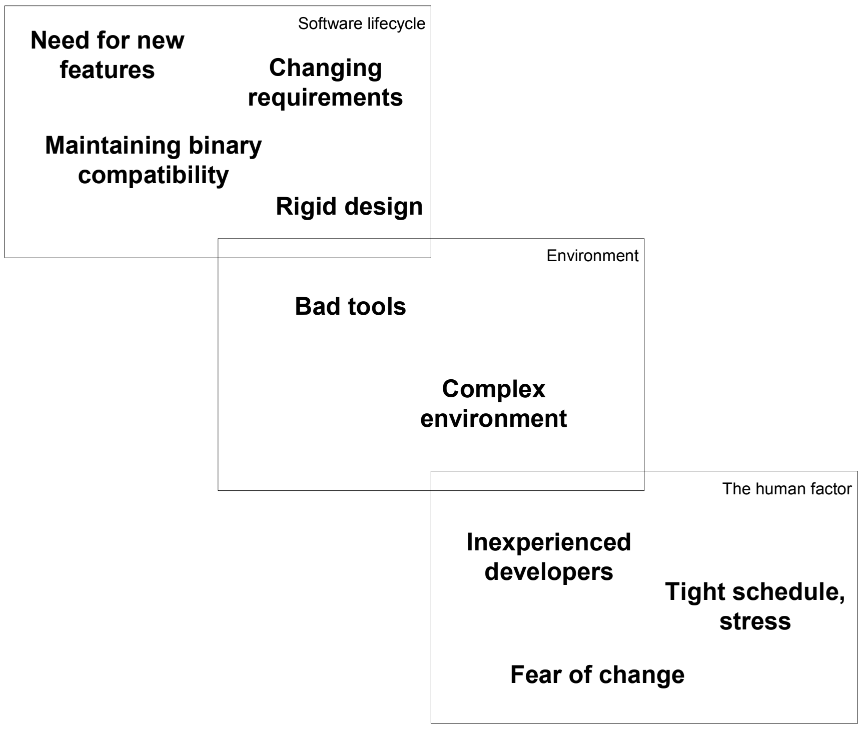


Figure 12. Reasons for code bloat categorised.

Need for new features. As time passes by an application usually is considered out-of-date and new features need to be added to attract e.g. new customers. The new features do not necessarily fit into the current architecture of the application. If the architecture and the module design are not rearranged to fit the new features they quickly become a mess. This is a very common situation and usually forces even a complete rewrite of many larger applications in every few years.

Changing requirements. It is a fact that software requirements do change. The customer wants an originally unplanned feature that does not quite well fit the original design. This causes workarounds and other more or less temporary solutions to be used. Little by little, the application fills with this kind of patchy code and becomes a nightmare to maintain. At the end, there are many similar code sections for handling nearly identical situations, distributed all over the application.

Maintaining binary compatibility. Over time, when platforms and applications evolve, it is often desirable to keep the public APIs constant at least in the sense of binary compatibility. Binary compatibility means that applications compiled with an older version of a DLL will continue to work with a new version. This effectively disallows any change to the existing API structure, even if the change would be absolutely required for e.g. better maintainability. Additions to the end of the API function list are still possible. This often leads to double or triple APIs containing the entire history burden as well as the new features.

Rigid design. Rigidity is the tendency of software to be difficult to change, even in simple ways /13/. Every change causes a cascade of subsequent changes in dependent modules. What begins as a simple two-day change to one module grows into a multi-week marathon of change in module after module as the programmers chase the thread of the change through the application. The biggest reason for rigidity is usually poor dependency management. Related to rigid design is v*iscosity* of the design. When faced with a change, engineers usually find more than one way to make the change. Some of the ways preserve the design, others do not (i.e. they are hacks). When the design preserving methods are harder to employ than the hacks, then the viscosity of the design is high. It is easy to do the wrong thing, but hard to do the right thing.

Bad tools. Symbian OS is a relatively young environment at least in its current, open, form. In addition, the developer community has been quite small until recently. The toolchain has not been really optimised for the task. The incorporated Cygnus GCC C++ compiler produces non-optimal machine code. For example, it does not take advantage of the ARM processor architecture's special features like conditional execution. Further still, there are no decent software based performance profilers or coverage analysis tools available for Symbian OS. Some expensive hardware based solutions do exist but those are not suitable for most of the developer community. There are some good compilers, profilers and other development tools available for ordinary Windows programming but they are just not suitable for the complex cross-compiling environment.

Complex environment. A modern object oriented operating system that is designed for mobile use is very complex and large environment. It contains so many specialist areas that even experienced developers do not remember all of the little details and quirks. Even just communicating with the network services requires knowledge of multiple protocols and techniques. This leads to non-optimal solutions and potentially prevents code re-use.

Inexperienced software developers. Not all developers are stellar programmers. Lack of knowledge about the platform and about the tools promotes the use of inferior methods, bad practices and overall inferior coding standards. This can be coupled with *Complex environment* above.

Tight schedule. Software is always done in a hurry. Time equals money and there is never enough money to do the perfect solutions. This forces programmers to take short cuts and use far-from-optimal algorithms and techniques because of lack of time for thinking and pondering.

Fear of change. "If it is not broken, do not fix it." mentality among programmers is very common. In other words, if an implementation works they are reluctant to change it even if the change would be a clear improvement. This leads to programmers adding new code when even a relatively small change in the existing code could have done the job.

To summarise, it can be said that usually many or most of the said reasons coexist in a software project. They also inhibit even strong interdependencies. One reason leads to another. A project may have a bad, rigid design, from the start. The project is then executed on a tight schedule with inexperienced software developers using bad tools on a complex environment. The customer wants new features that are added without proper designing. This leads to an even more rigid and viscose design. The rat race is ready.

4 METHODS FOR APPLICATION SIZE OPTIMISATION

This chapter describes methods for optimising the size of applications. Refactoring can reduce both the run-time and the on-disk footprint of the application. Compression usually reduces the on-disk footprint even considerably but may increase the run-time footprint momentarily. Most of the methods are not strictly Symbian OS specific but also a few such idioms are presented.

4.1 Refactoring

Refactoring is the process of changing existing software in order to improve its design and its implementation. Refactoring is risky. It requires changes to working code that can introduce hidden bugs. Refactoring becomes even riskier if it is practiced informally or ad hoc. It must always be coupled with thorough unit testing.

The refactoring patterns introduced later in this chapter concentrate on getting the size of the refactored applications down. There are many more refactoring patterns that do improve the design of software but make it larger meanwhile. Refactoring is the single most important form of size optimisation that can and must be done throughout the software lifetime.

4.1.1 Unit testing

Unit testing is a means of proving that a unit of software works as required by its specification /14/. Unit testing is usually seen as the testing of C++ classes as single entities. In unit testing classes are separated from the rest of the application and are tested in isolation. Communicating classes and interfaces are replaced with stubs, simulators or trusted entities.

There can be no reliable refactoring, or size optimisation, without unit testing. Unit tests ensure that the changes done do not affect the external functionality of the refactored software. Unit test cases must have sufficient coverage. At least all critical paths, memory allocations and sections that can cause exceptions must be tested.

Unit tests are best implemented and carried out with the help of a unit test framework. There exist no publicly available unit test frameworks for the Symbian OS. The implementation phase of this thesis utilises EUnit, an internal tool made at Digia /15/.

4.1.2 Patterns

Refactoring is characterised by the use of patterns. A pattern is a formal way of designing and implementing a common task /16/. Patterns are presented here by means of context, solution, example, and motivation. This chapter describes the following patterns:

- Decompose Conditional
- Extract Class
- Extract Method
- Move Field
- Move Method
- Pull Up Method

The presented patterns are identified from a large collection of publicly known patterns as those that most likely cause code size to shrink /7/. Some of the patterns are more concentrated on the actual implementation. Those are described in terms of C++ code examples. The rest of the patterns belong clearly to the design level. They are described with diagrams in Unified Modelling Language (*UML*) notation.

Decompose Conditional

Context:

- You have a complex conditional (if-then-else) statement.

Solution:

- Extract the condition into its own method.
- Extract the *then* part and the *else* part into their own methods.

Example:

Original

```
if( static_cast< CEikFloatingPointEditor* >( Control(
…EConverterEditor1 ) )->TextLength() == 0 )
   {
   iLastEditorUsedForUserInput = EConverterEditor1;
   iModel->SetSelectedEditor( 0 );
   ShowResultL();
   }
else if( static_cast< CEikFloatingPointEditor* >(
…Control( EConverterEditor2 ) )->TextLength() == 0 )
   {
   iLastEditorUsedForUserInput = EConverterEditor2;
   iModel->SetSelectedEditor( 1 );
   ShowResultL();
   }
```

Refactored

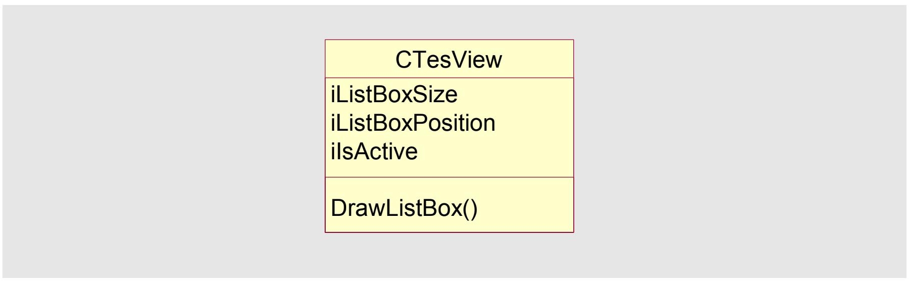
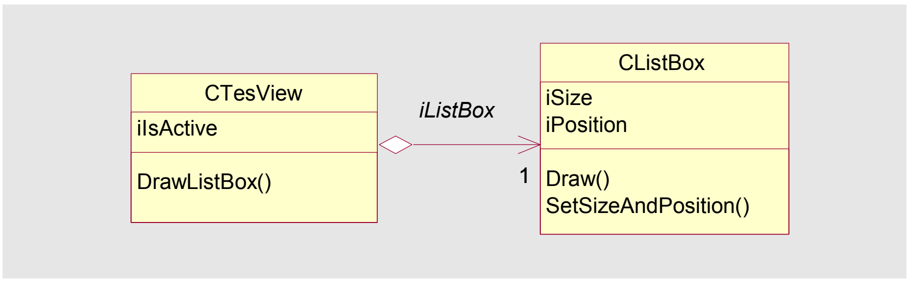
```
if( !IsFloatEntered( EConverterEditor1 ) )
   {
   SetLastEditor( EConverterEditor1 );
   }
else if( !IsFloatEntered( EConverterEditor2 ) )
   {
   SetLastEditor( EConverterEditor2 );
   }

TBool IsFloatEntered( TInt aEditorId )
   {
   ...
   }

void SetLastEditor( TInt aEditorId )
   {
   ...
   }
```

Motivation:

- Conditional logic is usually the most complex area of a program
- There is a good possibility that there are similar conditional statements elsewhere in the program and the extracted methods can be re-used there.

Extract Class	
Context:	• You have one class doing work that should be done by two. • You have a large class that tries to do everything.
Solution:	• Decide how to split the responsibilities of the class. • Create a new class and move the relevant fields and methods from the old class into the new class. • If the original class is part of a public interface, you have to provide wrapper methods in place of the moved methods.
Example:	**Original**  CTesView iListBoxSize iListBoxPosition iIsActive DrawListBox() **Refactored**  CTesView iIsActive DrawListBox() iListBox 1 CListBox iSize iPosition Draw() SetSizeAndPosition()
Motivation:	• A large class is too big to understand easily. The lack of understanding causes accelerated growth when the class is modified. • Clearly defined classes tend to stay small and are easier to maintain.

Extract Method

Context:

- You have a code fragment that can be grouped together.
- Copy-paste coding. The same fragment shows in multiple methods.

Solution:

- Turn the fragment into a method whose name explains the purpose of the method.

Example:

Original

```
void FetchFirstItemL( TPtr& aName, TReal& aValue )
   {
   TInt index = DoSomeImportantStuff();
   aName = view.ColDes( index++ );
   aValue = view.ColReal( index++ );
   }

void FetchOtherItemL( TPtr& aName, TReal& aValue )
   {
   DoSomethingElse();
   aName = view.ColDes( 0 );
   aValue = view.ColReal( 1 );
   }
```

Refactored

```
void FetchFirstItemL( TPtr& aName, TReal& aValue )
   {
   TInt index = DoSomeImportantStuff();
   FetchNameAndValue( aName, aValue, index );
   }

void FetchOtherItemL( TPtr& aName, TReal& aValue )
   {
   DoSomethingElse();
   FetchNameAndValue( aName, aValue, 0 );
   }

void FetchNameAndValue( TPtr& aName, TReal& aValue,
TInt aColumnIndex )
   {
   aName = view.ColDes( aColumnIndex++ );
   aValue = view.ColReal( aColumnIndex++ );
   }
```

Motivation:

- Copy-paste coding is one of the worst cases of code bloat.
- Because the same code is copied to several places, all modifications to the code fragment must also be done several times. Missing one fragment causes a potential bug.
- Any existing bugs will also be copied along.

Move Field	
Context:	• A field is, or will be, used by another class more than the class on which it is defined.
Solution:	• Create a new field in the target class, and change all its users. • Move also the getter and setter methods for that field where appropriate (see *Move Method* pattern).

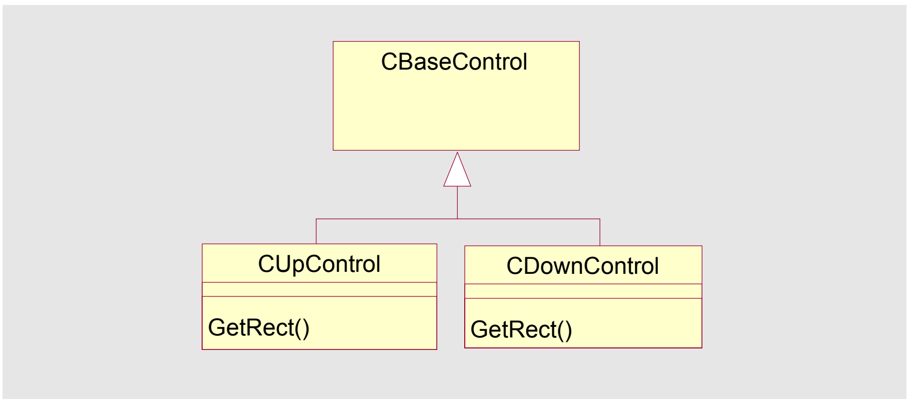
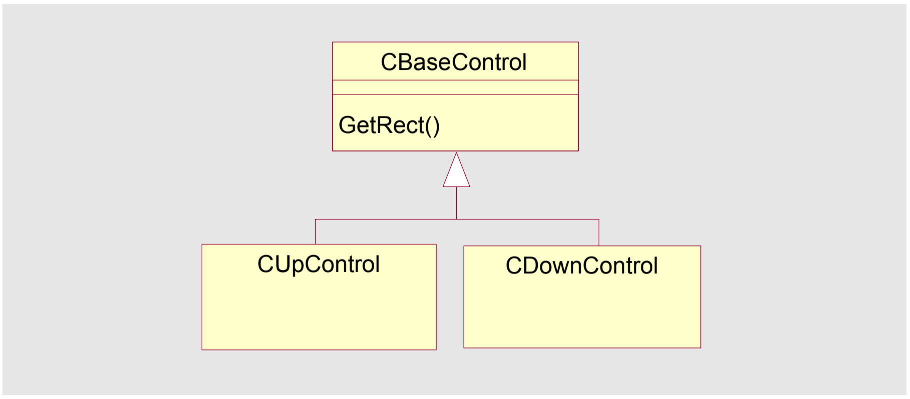
Example:	**Original**
	 `class CUiView`
	`   : public CCoeControl`
	`   {`
	`   public: // Methods`
	`      ...`
	`   }`
	
	`void CUiView::Draw() const`
	`   {`
	`   TSize mySize = iParent->ViewSize();`
	`   TPoint myPos = iParent->ViewPosition();`
	`   TRgb myColour = iParent->ColourMap()->ViewColour();`
	`   gc.FillRect( myPos, mySize, myColour );`
	`   }`
	
	 Refactored
	 `class CUiView`
	`   : public CCoeControl`
	`   {`
	`   public: // Methods`
	`      ...`
	
	`   private: // data`
	`      TSize iSize;`
	`      TPoint iPoint;`
	`      TRgb iColour;`
	`   }`
	
	`void CUiView::Draw() const`
	`   {`
	`   gc.FillRect( iPos, iSize, iColour );`
	`   }`

Motivation:	• "Unnecessary" getter and setter method calls cause overhead that shows in performance and on the binary size. • A field in the "wrong" class causes many dependencies that could be avoided.

Move Method	
Context:	• A method is, or will be, using or used by more features of another class than the class on which it is defined
Solution:	• Create a new method with a similar body in the class it uses most. • Either change the old method into a delegation, or remove it altogether.

Example:	**Original**
	`void CUiControl::Draw() const`
	`   {`
	`   TRect rect = iBox->Rect();`
	`   TRgb fgColour = iBox->FgColour();`
	`   TRgb bgColour = iBox->BgColour();`
	`   gc.FillRect( rect, fgColour, bgColour );`
	`   }`
	
	Refactored
	`void CUiControl::Draw() const`
	`   {`
	`   iBox->Draw();`
	`   }`
	
	`void CBox::Draw() const`
	`   {`
	`   gc.FillRect( iRect, iFgColour, iBgColour );`
	`   }`

Motivation:	• Moving methods is necessary when classes have too much behaviour (i.e. are too large) or when classes are too tightly coupled. • Removing method calls that are required to access, for example, private member variables of another class, reduces the binary size. Method calls are always more expensive than member variable access in terms of generated machine code size and software performance.

Pull Up Method	
Context:	• You have methods with identical results on subclasses.
Solution:	• Move them to the base class.
Example:	**Original**  CBaseControl CUpControl GetRect() CDownControl GetRect() **Refactored** 
Motivation:	• Eliminating duplicate behaviour is important. It reduces code bloat and increases maintainability. • Making modifications to similar code in several places risks that some of the places are missed and potential bugs are introduced.

4.2 Compression

Compression is the procedure where an input data stream is converted into another data stream that is smaller in size /17/. A stream is either a file or a buffer in memory. The compressed data stream must be decompressed before use, as it is nearly always unusable in the compressed form. Compression is an easy and quick form of size optimisation affecting the application's on-disk footprint.

Compression methods can be roughly divided into two categories: *lossy* and *lossless compression* /17/. Lossy compression methods achieve better compression by selectively throwing away some information. When the compressed data stream is decompressed, the result is not identical to the original data stream. Such a method suits especially well into compressing images, movies or sounds. If the loss of data is relatively small, the human eye may not be able to tell the difference. In contrast, content sensitive files (e.g. user documents, program binaries), may become worthless if even a single bit is modified. Such files must be compressed only by lossless compression methods. Table 3 shows which compression method suits each data type. The named methods are described in more detail in the following chapters.

Data type	Compression method(s)	Compression type
Code	Thumb, LZ	Lossless
Text, resources	LZ	Lossless
Graphics	DCT, LZW	Lossy, Lossless
Audio	A/μ-Law, PCM	Lossy

Table 3. Compression methods.

4.2.1 Code

ARM Thumb architecture. The easiest and cheapest way to reduce the size of executable code is to use the ARM Thumb instruction set. An overview of the architecture was presented in 2.2.1. The Thumb instruction set can be seen as a compressed form of a subset of the ARM instruction set. Thumb instructions map onto ARM instructions, and the Thumb programmer's model maps onto the ARM programmer's model. The Thumb mode uses on-the-fly decompression in an ARM instruction pipeline and then instructions execute as standard ARM instructions within the processor.

The resulting compression ratio and code performance varies from application to application, but on average the properties are as follows /6/:

- Thumb code requires 70% of the space of ARM code.
- Thumb code uses 40% more instructions than ARM code.
- ARM code is 40% faster than Thumb code (with a 32-bit memory bus).

Thumb is fully supported by the Symbian Platform development tools, and an application can mix ARM and Thumb code modules flexibly to optimise performance or code density. Application developers do not need to known beforehand if their application will be compiled into ARM or Thumb code at the end. The resulting binary format is defined by compiler parameters and can be changed at any time.

Executable module compression. Binary code modules can also be compressed, for example, using Lempel Ziv (*LZ*) derived methods. The code responsible for loading the modules must then be able to decompress them before execution. It should also be possible to construct self-decompressing executables. This is recognised as an important subject but is not discussed in more detail due to its complexity. It must also be noted that executable compression does not provide any benefits with modules that are executed straight out from ROM without loading them into RAM first.

4.2.2 Text, resources

Text files, resource files and other program data files must be compressed with a lossless compression method. Symbian OS contains an implementation of the ZLib compression library /18/. ZLib is based on a variant of the LZ77 algorithm called *deflation*. The maximum compression factor is theoretically 1032:1. It should be noted that this level of compression is extremely rare and only occurs with trivial files (e.g. a megabyte of zeros). Usual ZLib compression ratios are on the order of 2:1 to 5:1.

If one wants to access data for example at the end of a compressed file, the whole file needs to be decompressed. This requires extra processing power and memory. With a large file this might not even be possible at all. A solution to this problem is to implement a file layer that enables random access of a compressed file without compressing and decompressing the entire file /19/. There is no such layer ready-made in Symbian OS.

4.2.3 Graphics

The Symbian OS Font and Bitmap server provides native support for Run Length Encoding (*RLE*) based Multi-Bitmap (*MBM*) image format. Industry standard Discrete Cosine Transform (*DCT*) based JPEG and Lempel Ziv Welch (*LZW*) based GIF are supported by the Media Server /1/.

The de-facto image file format on the Symbian Platform is the MBM format. It provides support for multiple bit-depths and is able to store more than one image in one file. It is rather handy to use from the programmers viewpoint but it is not supported by any graphics production software and must be created from Windows Bitmap originals by a converter tool. MBM uses RLE compression to reduce the size of the files. Because the RLE compression is a lossless method, it produces mediocre compression ratios. Therefore, it is advisable to utilise lossy compression methods for larger images.

Table 4 shows the comparison of a 200 x 200 pixels sized image saved in JPEG, GIF and MBM formats. The images clearly show that JPEG or GIF should be used instead of MBM when dealing with larger images such as the ones below. MBM should still be used for tiny icons in the size range of 8 x 8 to 32 x 32 pixels. Notice the large variances in image file sizes and colour qualities.

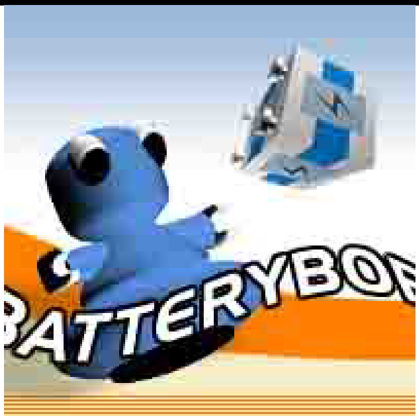
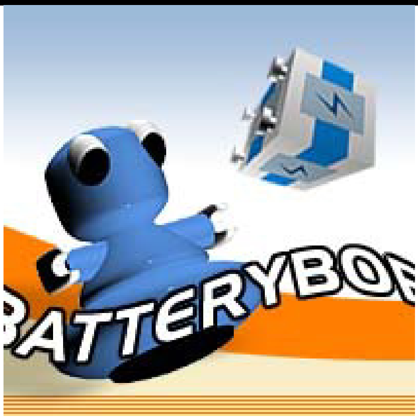
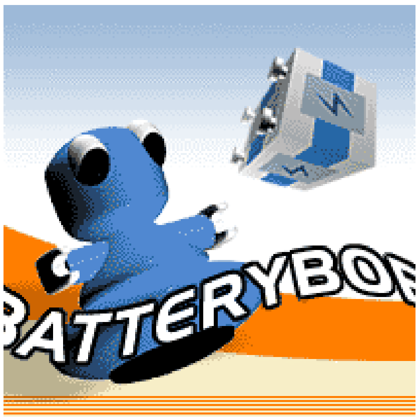
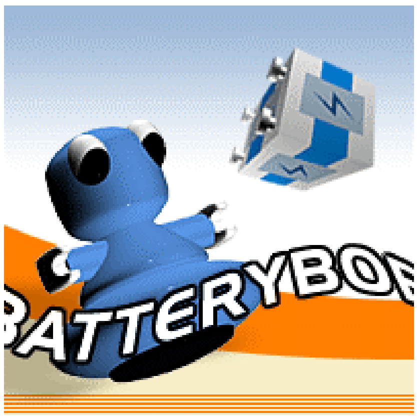
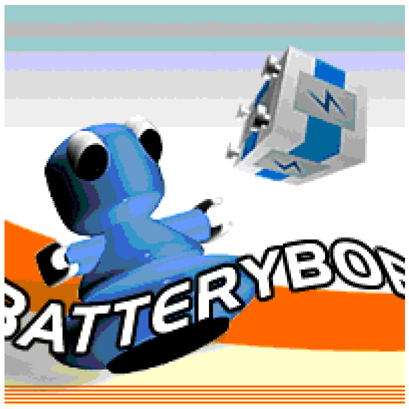
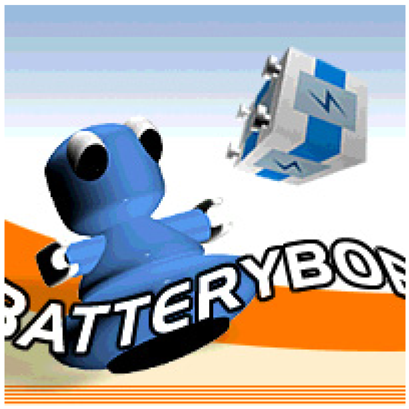
Low-colour or low-quality	High-colour or high-quality
 JPEG at low quality, 5 329 bytes	 JPEG at medium quality, 9 577 bytes
 GIF at 32 colours, 8 737 bytes	 GIF at 256 colours, 14 378 bytes
 MBM at 256 colours, 11 905 bytes	 MBM at 4096 colours, 19 334 bytes

Table 4. Comparison of JPEG, GIF and MBM compressed images.

4.2.4 Audio

The Symbian OS Media Server is able to play audio files stored in the popular WAV and Sun AU file formats. Developers can add support for other formats if they have access to the target device's hardware drivers. This usually requires an agreement with the manufacturer.

The WAV format as such allows the use of compressing codecs but the Symbian implementation supports only non-compressed Pulse Code Modulation (*PCM*) encoded audio. Each PCM encoded sample takes eight bits of space. Therefore, one second of sound at the sample rate of 8000 Hz takes 64 000 bits, or 8000 bytes.

AU files are usually processed with either A-Law or µ-Law companding. Those techniques are originally used in Integrated Services Digital Network (*ISDN*) digital telephony. µ-Law in North American and Japan, and the A-Law elsewhere. They represent 14-bit or 13-bit samples as 8-bit samples. This leads to an identical file size compared to PCM but with a better sound quality and higher dynamics. However, as the current Symbian OS devices inhibit inferior sound output capabilities there is little benefit in using A-Law or µ-Law at the moment.

4.3 Symbian OS idioms

This chapter describes several idioms specific to Symbian OS software development and software running on the OS. They affect mostly the run-time footprint of software. Memory consumption can increase even considerably if they are used incorrectly.

4.3.1 Exception handling

Applications must perform proper cleanup when an exception occurs, because they are designed to run for long periods (months or even years) without interruption or system re-boot. They must not slowly leak memory under any circumstances. The operating system will do the cleanup when the application has been closed, but before that, it is the application's responsibility.

Trap harness. Symbian OS does not use the standard C++ exceptions (try-catch blocks), but supplies its own methods. This is because C++ exceptions have a relatively high memory overhead. An exception is referred to as a *leave*. A trap harness defines a point in program code that will be immediately jumped to if a leave occurs. Leaves are raised through calling functions provided for the purpose by the system. Many system functions can result in leaves. Leaves may also be raised in user code.

Cleanup stack. When a leave occurs, any heap allocated resources referred to only through automatic pointer variables will be orphaned on the heap, effectively causing a memory leak. To prevent leaks, it is necessary for the program to keep a handle to such objects, so that the system can automatically find and clean them up. The cleanup stack is the means by which this is done. GUI applications have a cleanup stack supplied to their main thread by the application framework. Other threads and other applications must explicitly create the cleanup stack object by themselves.

Trap harness and cleanup stack work as a pair. When a leave occurs a trap harness is responsible for deleting all the objects that have been placed in the cleanup stack inside the current harness. There must be at least one trap harness in each thread or the exception handling system will not function.

4.3.2 Descriptors

Symbian OS does not contain a specialised string class. Instead, it provides general-purpose buffer objects known as *descriptors*. Descriptors provide a safe, consistent and economical mechanism for accessing and manipulating strings and general binary data. A descriptor represents data that can reside in any memory location, either ROM or RAM. A descriptor object maintains pointer and length information to describe the data. All access to the data is made through the descriptor object. There are two base classes for descriptors: one for read-only descriptors and one for modifiable descriptors.

Operations on data represented by a descriptor are safe. Accidental or deliberate attempts to access memory outside the data area represented by a descriptor are caught. An illegal access is treated as bad programming rather than an environment or resource problem and raises an exception known as a *panic*.

The concrete descriptors that a programmer uses are categorised into three types:

- *Buffer descriptors* - where the data is part of the descriptor object and the descriptor object lives on the program stack. Implemented in `TBuf` and variant classes.
- *Heap descriptors* - where the data is part of the descriptor object and the descriptor object lives on the heap. Implemented in `HBuf` and variant classes.
- *Pointer descriptors* - where the descriptor object is separate from the data it represents. Implemented in `TPtr` and variant classes.

An application developer must remember a couple of important aspects about descriptors. First, stack-based descriptors should always be passed to functions 'by-reference'. This means that only a pointer or a reference to the descriptor is given as a function parameter. If such a descriptor is passed 'by-value', a copy of the data contained in the descriptor is automatically created by the C++ compiler. After the function call returns the temporary copy is discarded. It is not widely known that the Symbian OS *delete* operation fills the freed memory block with zeros. This wastes memory and precious processor cycles. The same is actually true for all other objects as well.

Second, descriptors should be created with an optimal length. This is especially important with 16-bit Unicode descriptors. For example, creating a `TFileName` descriptor for storing a file's name and path consumes 512 bytes of memory even if the actual stored name is 32 characters long. This is because the descriptor has a fixed size of 256 characters. One character takes two bytes on a Unicode build (usually all builds are Unicode).

4.3.3 Thin templates

C++ templates can be difficult to manage. A template really defines a whole family of classes. Each member of that family that is instantiated requires its own code. Code duplication must be avoided at all costs. The solution is to use *thin templates*.

The thin template idiom begins with a base class which has code in terms of `TAny*` parameters:

```
class CArrayFixBase ... {
  IMPORT_C const TAny* At( TInt aIndex ) const;
```

This base class has the real code, just once. The code resides in a single DLL and is exported from that DLL. The base class may be fat, i.e. it may contain an arbitrary amount of code. Then, a derived class is constructed as follows:

```
class CArrayFix< T >
  : public CarrayFixBase
  {
  inline const T& At( TInt aIndex ) const
    {
    return *( const_cast<const T*>( CArrayFixBase::At( anIndex ) ) );
    }
```

Because this class uses only inline functions, it generates no extra code. However, because the casting is encapsulated in the inline function, the class is typesafe to its users. In addition, C++ compilers usually do not generate template code that is not used.

4.3.4 Emulator vs. hardware

It is very easy to develop an application that runs happily on the emulator but fails miserably after integration to hardware. Inexperienced developers often do not realise that on the target hardware their application may have very little memory available. Memory capacity is far greater on the emulator by default. It can be set by a configuration file settings. It is highly recommendable to change the default value of 16 megabytes to 4 megabytes, or whatever matches the particular target hardware best.

The performance, or speed, of the software running on the emulator has nothing to do with the actual target device performance. The emulator does not even try to mimic any particular device. Tuning software performance requires always testing on the real hardware platform. Even a very bloated application may appear to be working fine until it is tested on hardware.

The Symbian OS is a multi-process, multithreaded system. However, the emulator is implemented in such a way that the kernel and all the applications run under a single Windows process. Native Symbian OS processes and threads are represented as Windows threads. This may cause problems if the application developer implements Inter-Process Communication (*IPC*) in a questionable way. For example, the developer might try to access other processes' memory space. This works on the emulator but fails on hardware because processes are not allowed to do that in the "real world".

4.3.5 Front End Processors

A Front End Processor (*FEP*) is a component that is loaded automatically with all GUI applications. It sits on top of every application intercepting all input from the user. It is then able to process the input and forward it to the underlying application the way it wishes. The most notable application for FEPs are virtual keyboards and handwriting-recognition software for touch-screen devices.

For example, let's assume that a single FEP consumes about 200 kilobytes of RAM when executed. In an everyday situation there are four applications running: desktop, application launcher, file manager and word processor. The net effect of the loaded FEPs, one in each application, is 800 kilobytes. If this FEP would be installed in the Nokia 9210, the execution memory would fill up in no time. Therefore, it is important to implement FEPs in such a manner that the instances stay small. This can be accomplished by implementing the core logic as a *server* and the FEP instances as *clients* for the server.

5 OPTIMISATION AND THE SOFTWARE PROCESS

This chapter introduces a real-world software process model. A process for planning and implementing size optimisation is also described and integrated into the standard process model. The primary goal of the optimisation process is to get the software size down at once. The secondary goal is to restrict and control the growth of the software over time.

5.1 Digia Software Process lifecycle

The Digia Software Process (*DSP*) is an incremental and iterative software process model. Figure 13 shows the two major sets of phases and activities during the lifecycle: planning and construction. The same model is used for both customer projects and internal product development. The pictures in this chapter are originally from the DSP Process Introduction presentation /20/.

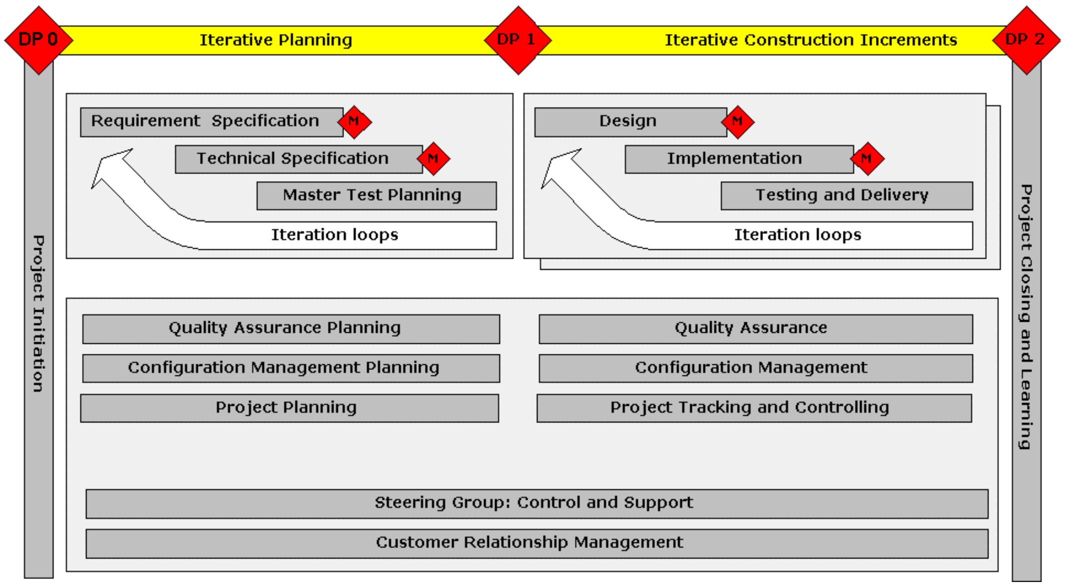

DP 0
Iterative Planning
DP 1
Iterative Construction Increments
DP 2
Project Initiation
Requirement Specification
M
Technical Specification
M
Master Test Planning
Iteration loops
Design
M
Implementation
M
Testing and Delivery
Iteration loops
Project Closing and Learning
Quality Assurance Planning
Configuration Management Planning
Project Planning
Quality Assurance
Configuration Management
Project Tracking and Controlling
Steering Group: Control and Support
Customer Relationship Management

Figure 13. Digia Software Process overview.

The planning phase has only one increment whereas the software construction phase is divided into two to five increments. Several iterations of the predefined activities are done inside a single increment. An increment's timeframe may range from one to six months, depending on the project. The construction increment is shown in Figure 14 with more detail.

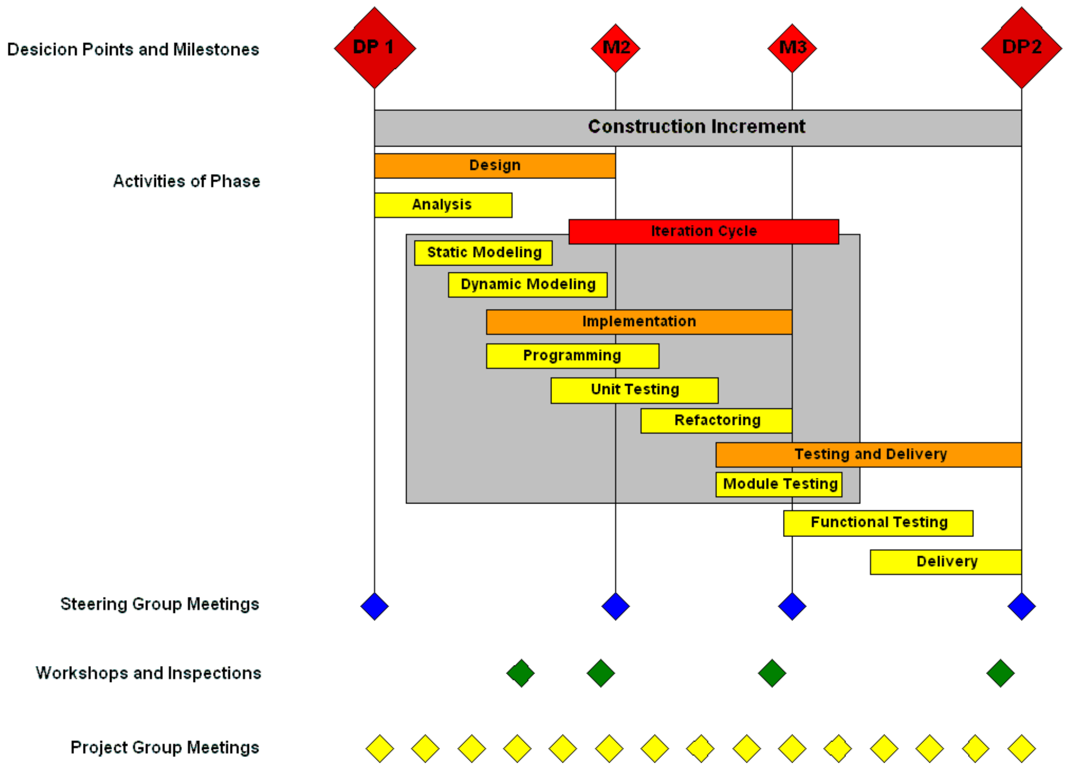


Figure 14. DSP Construction Increment overview.

Design phase. Internal implementation of a specific module and interface is specified. Module design describes why and how certain design and implementation solutions are used. The design phase produces a design specification, a test specification and the code skeleton for the module.

Implementation phase. The design is implemented using language and environment dependant solutions. Unit testing is used to verify the functionality of the module's sub-units. Conformance to coding conventions is also checked. The implementation phase produces source code, binary files and data.

Testing and delivery phase. The application produced in the implementation phase is tested and checked against its requirements. It is also ensured that the application is built and delivered as agreed with the customer. The testing and delivery phase produces entries to the defect database, test reports and delivery packages.

5.2 Optimisation process

There are basically two kinds of optimisation cases. The first kind is the optimisation of an existing application that has been discovered too large after its completion. These are increasingly common cases nowadays. The second kind is the optimisation of an application throughout its development lifecycle. These must be far more common in the future.

5.2.1 Existing application

The optimisation is done utilising a process that is separate from the standard software implementation process and is done afterwards. The phases 2 to 5 can be repeated in an incremental fashion for fine-grained control.

The phases are defined as follows:

1. Static and dynamic analysis.
2. Optimisation planning.
3. Optimisation implementation.
4. Functional testing.
5. Measurement.

The individual phases and their relative positions are illustrated in Figure 15.

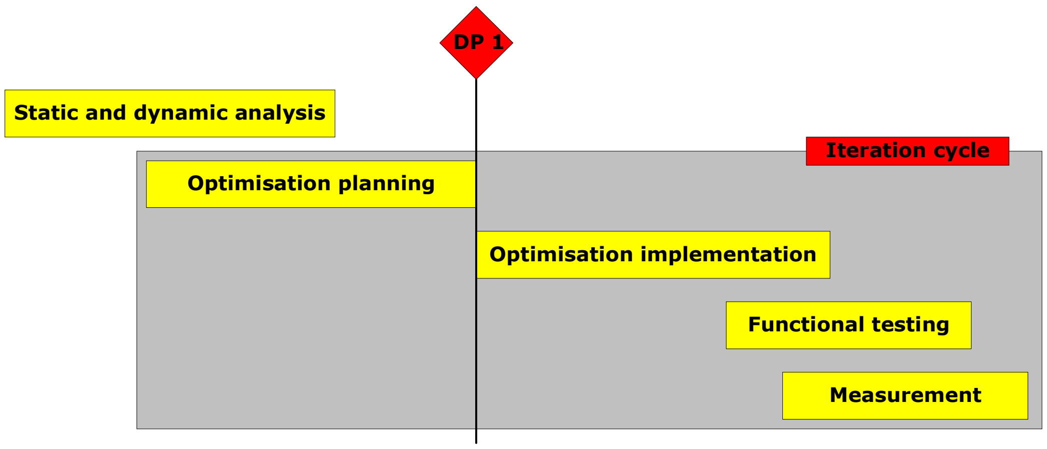


Figure 15. Optimisation process phases.

Static and dynamic analysis. All the files are checked, including any documentation. Static analysis is done by going through the documentation and looking at the source code. It can be supported by automated tools that calculate metrics based on e.g. lines of code per class, number of dependencies between classes and so forth. Tools that visualise the static class hierarchy and relationships are also valuable. Dynamic analysis is done with profiler tools measuring memory consumption, execution paths and execution timing. This phase produces an analysis report.

Optimisation planning. Targets for the optimisation are set. Based on the targets and the analysis results from the previous phase, methods for the optimisation are evaluated and selected. At this point (DP 1), it is crucial to decide if the optimisation should be implemented or not. The cases where the latter is true are common and time should not be wasted in trying to optimise something that is not really optimisable. However, this requires quite a lot of expertise from the analysts. This phase produces an optimisation plan.

Optimisation implementation. The planned optimisations are implemented. Data files are compressed where needed and uncompressing code is added to the application. Code refactoring is done. Unit and module tests are done wherever possible. The tests have a more important role than functional testing because they can be done automatically by specialised tools. Functional testing requires human labour and therefore is more expensive and time consuming. This phase produces source code and binary files.

Functional testing. The application is tested according to its functional test specification. If the application does not have a complete test specification, it must be created and executed once before any of the optimisation phases are done. The testing phase provides a means of proving that the optimised application inhibits the same visible functionality as before optimisation. This phase produces entries to the defect database and test reports.

Measurement. The resulting application is measured both statically and dynamically using the same methodology as in the first phase but concentrating on the modified parts. The achieved results are compared to the targets set in the second phase. This phase produces an optimisation summary.

5.2.2 New application

The activities of the process phases described above in 5.2.1 are integrated to the standard software process. Optimisation related work should be done as a part of the refactoring activities of a phase. Refactoring patterns described in 4.1.2 can be successfully employed there. The aspects of software that affect the resulting footprint must be carefully decided in technical architecture specification while in the planning phase. For example, it is to be decided where to use compression and where data is to be left as is. Compression techniques are discussed in 4.2. The complete or near complete rewriting of an application follows the same rules.

6 CASE STUDY

A real application was selected for studying and implementing the optimisation methods described in this thesis. The application has been made by a third party, but has been given to Digia for improvement and maintenance.

The application in question is an end-user application that is able to convert all kinds of units to other units. For example, the user can check how many kilometres is three miles. This scheme should be very straightforward and easy to implement. Obviously, in this particular case, it has not been. The application has been implemented for a platform that is based on the Symbian OS version 6.1.

6.1 Analysis

The analysis was done in three steps. First, the application was compiled and the resulting file sizes were measured. Second, the documentation was gone through in order to understand the requirements and the structure of the implementation. Finally, the source code was reviewed in detail.

6.1.1 Measurement

The application consists of several files that belong to the two modules: UI and engine. The engine module uses a relational database to store its data. The size of the database varies over time without clear logic. The size was observed to reach over 50 kilobytes at times. An estimated average value of 30 kilobytes was chosen for the results. The individual files and their sizes are shown in Table 5.

File	Type	THUMB (bytes)	ARMI (bytes)
UI module	.app	21 516	24 536
UI resource file	.app rsc	2 014	1 956
UI settings file	.ini	285	285
Engine module	.dll	12 332	13 556
Engine resource file	.dll rsc	2 098	2 088
Engine database file	.db	30 720	30 720
Application info file	.aif	2 516	2 516
	TOTAL SIZE:	**71 481**	**75 657**

Table 5. Application file sizes on THUMB and ARMI.

It is important to notice that the database takes quite a lot of the total size and therefore could be one of the first targets for optimisation. The relative file sizes resulting from a THUMB build are illustrated in Figure 16.

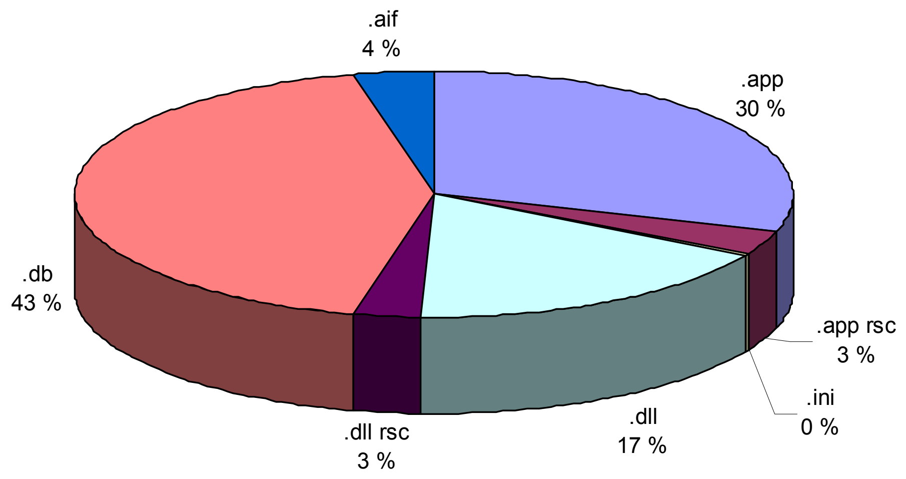


Figure 16. Relative binary file sizes.

The application contains seven classes, of which six are located in the UI module and one in the engine module. The number of lines of code per class is 713 on average, but there is a lot of variance from class to class. The engine module has 17 exported methods, which is quite a lot for one class. The application size in lines of code is shown in Table 6.

Module	Class name	Lines of code	Exports
UI:			
	CConverterApplication	91	1
	CConverterDocument	159	-
	CConverterAppUi	195	-
	CRatesForm	582	-
	CConverterDbModel	1 037	-
	CConverterMainForm	1 224	-
Engine:			
	CConverterDB	1 702	17
	TOTAL:	**4 990**	**18**

Table 6. Application size in lines of code.

The relative sizes of individual classes are illustrated in Figure 17.

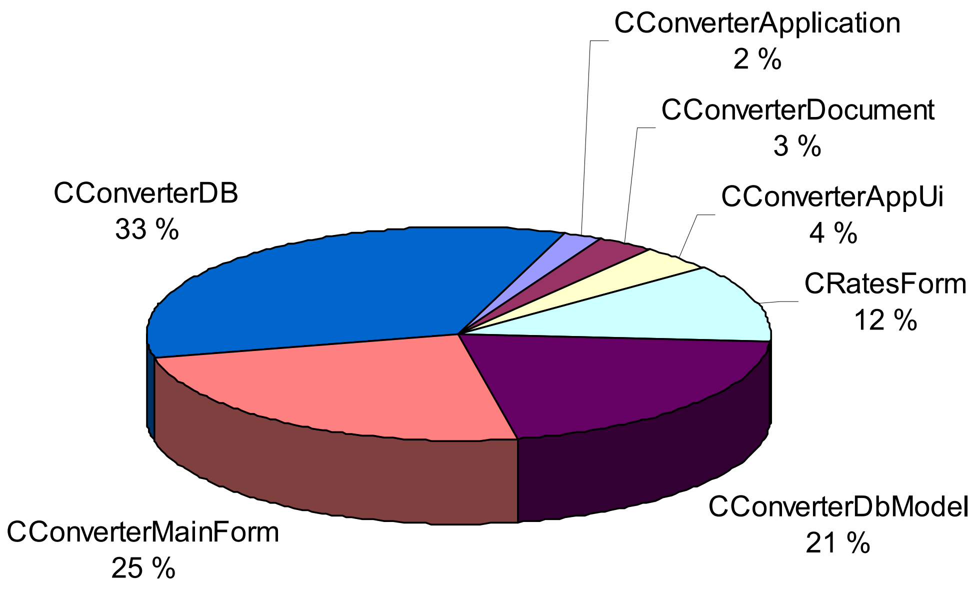


Figure 17. Relative class sizes in lines of code.

6.1.2 Documentation check

There exist exactly two documents about the application. The first one is the user interface functional specification. The second one is the design document for the application engine's base class. The UI specification seems to be adequately updated and is in good shape. The engine design document has not been updated for over a year and clearly does not reflect the current state of the implementation.

As there was no UML class diagram available that would cover the UI module, a reverse engineering phase was necessary. Class diagrams are an essential way of getting an overall understanding of the application's structure. Based on automatic source code analysis, a class diagram was created using Rational Rose /21/ software. The resulting class diagram is shown in Figure 18.

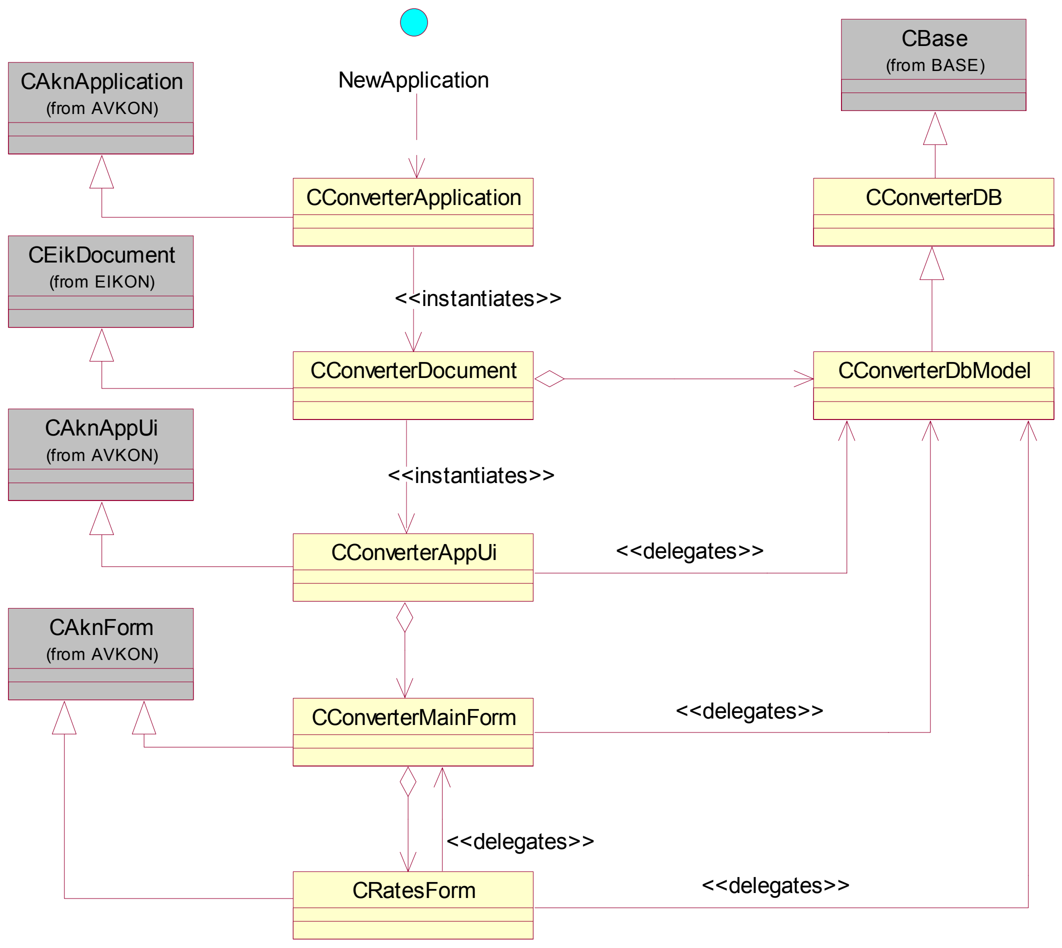


Figure 18. Application class diagram.

In addition to the dependencies shown in the diagram, there are probably many hidden dependencies between the classes that can be found only be closely inspecting the source code.

6.1.3 Decision

It was decided that refactoring this particular application is not feasible. This is primarily because of the following reasons together.

Monolithic design. The core application logic is practically implemented in three huge classes (`CConverterDB`, `CconverterDbModel and CConverterMainForm`). In addition, first two of the classes have an 'is-a' relationship: the second class is derived from the first one. Program logic is scattered throughout the three classes. It looks like someone had used a shotgun to decide what goes where.

Lack of testability. The application has a largely monolithic design. One cannot possibly write unit test cases for a 'unit' that means half of the application. That would be module testing. However, module testing is not practical either in this case. This is because the engine module and the UI module are so tightly coupled. Refactoring cannot be done without proper unit testing.

Lack of documentation. The documentation is very inadequate. The inner application logic has not been documented at all. It is not practical to try to guess all the details by just looking at the source code. In addition, the original authors of the application are not available for questioning.

Magic numbers. There are many hard-coded number values in the code without reasonable comments. Most of the implementation is based on passing undocumented table indexes and other discrete values around the function calls. There are even multiple functions just for converting magic numbers to other forms of magic numbers. It is not possible to guess the meaning of all the values reliably.

The application will be written from scratch while trying to reuse as many small fragments of the UI module code as possible. Most of the existing UI resource definitions should also be usable straight away.

6.2 Implementation

While and after the implementation of the case study observations about good and bad practices were noted and examined. The following two chapters describe the findings.

6.2.1 Observations

Inheritance and virtual functions. In C++, deriving a class from another class introduces a Virtual Function Table (*vftable*) to the binary module. A vftable is used to direct function calls made via the base class' virtual functions to the derived class. One vftable is created for each class in the derivation chain. Deriving a class from a deep class hierarchy means creating multiple vftables simultaneously. The result is increased binary size. For example, deriving a class from `CAknAppUi` (instead of not deriving at all), 2532 bytes worth of vftables and lots of external dependencies are introduced to the binary. The large size increase is explained by the deep class hierarchy above `CAknAppUi` and its high number of DLL dependencies.

Code re-use. Re-using existing modules or even single classes can be tricky. Classes are usually not designed for use outside the original scope. They may have many dependencies to other classes and therefore cannot be used without them. A far easier alternative to module or class re-use is code snippet re-use. The rewriting of an application can be greatly shortened by copying small fragments of code from the original implementation. This is mostly true for the parts of code that deal with UI components and the UI framework in general. However, great care must be taken as not to copy any existing bugs along.

Relational databases. A database always has more or less overhead compared to raw data storage. Database Management Systems (*DBMS*) need to store additional information about the data structures such as tables, views and indexes. The Symbian

OS DBMS uses the industry standard Structured Query Language (*SQL*) for database access. An application that uses a database has to create SQL query strings and receive data from the DBMS. It has to have sufficient memory buffers for the operations, which can consume considerable amounts of memory and cause obscure out-of-memory errors.

The DBMS has to read and write the data to permanent storage, which usually resides on a Flash file system. Flash memory is slow to read and write, and causes serious performance degradation. The original case study application had a start time of over twelve seconds just because it was initialising a database. The rewritten implementation, that uses a resource file for data storage, starts in less than a second.

Memory strategy. A memory strategy defines the points in which a module allocates memory, i.e. it defines the dynamic part of the run-time footprint. There are three basic strategies /22/. In *static strategy* all the needed memory is allocated in the beginning. In *dynamic strategy* most memory is allocated in the beginning and the least important parts later. Partial failure must be implemented. In *random strategy* memory is allocated and freed throughout the lifetime. Static strategy is very straightforward to test for out-of-memory errors, dynamic strategy is somewhat harder, and random strategy it is next to impossible (for 100% test coverage).

The data model (engine module) of the implemented application uses static memory strategy. It allocates all the needed memory at construction and thus cannot possibly run out of memory during its lifetime.

6.2.2 Recommendations

Based on the described observations, a few recommendations are be made.

1. Favour aggregation over inheritance ('has-a' relationship, not 'is-a'). Especially deriving from the UI framework classes should be avoided wherever possible.

2. If you are allowed to rewrite an application, do so. However, use the original implementation as a base for up to 50% shorter development process.
3. Do not use databases in mobile software. Databases are large and carry a significant performance hit.
4. Choose your memory strategy wisely and make your customer accept it. No memory strategy means a random strategy.

6.3 Results

The on-disk footprint of the created application is about one third of the original application size. The most visible reason for that is the abandonment of the relational database that was used by the original authors. With the data set size (a couple hundred, mostly static, entities) of this application such an implementation was considered largely overkill. Approximately 80% of the resource file contents could be reused with little or no modification. Practically, only a little cleanup and restructuring was needed.

6.3.1 Measurement

The individual files and their sizes are shown in Table 7. The measures of the new application are also compared to the old ones.

File	Type	Old size (bytes)	New size (bytes)	Change (bytes)
UI module	.app	21 516	13 100	-8 416
UI resource file	.app rsc	2 014	4 678	+2 664
UI settings file	.ini	285	779	+494
Engine module	.dll	12 332	4 132	-8 200
Engine resource file	.dll rsc	2 098	0	-2 098
Engine database file	.db	30 720	0	-30 720
Application info file	.aif	2 516	2 516	+0
	TOTAL SIZE:	**71 481**	**25 205**	**-46 276**

Table 7. Application file size differences.

The new application has nine classes, of which five are located in the UI module and four in the engine module. The number of lines of code per class is 253 on average and is more equally spread compared to the old application. The application size in lines of code is shown in Table 8.

Module	Class name	Lines of code	Exports
UI:			
	CCnvApplication	85	1
	CCnvDocument	94	-
	CCnvAppUi	112	-
	CCnvMainForm	641	-
	CCnvRatesForm	369	-
Engine:			
	CCnvConverter	227	8
	CCnvCategory	303	-
	TCnvCategory	299	19
	TCnvUnit	160	-
	TOTAL:	**2 290**	**28**

Table 8. Rewritten application size in lines of code.

The relative sizes of individual classes are illustrated in Figure 19.

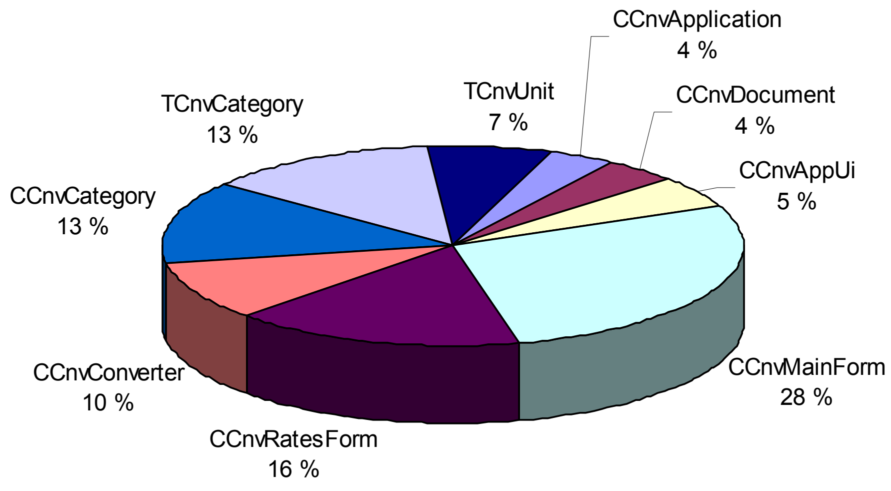


Figure 19. Relative class sizes in the rewritten application.

6.3.2 Inheritance and binary size

A few additional tests were made to get more detailed knowledge about the factors involved in inheritance (in C++). In order to understand the test and its results, one most likely has to have at least some Symbian OS software development experience. The case study application engine was used as a placeholder for a couple of test classes. The test classes were identical (except the name, of course) and were not derived from any class. The tests were made according to the following process:

1. The module was compiled with only one of the new classes present and the resulting binary size was measured for reference (*).
2. A single new class was changed to be derived from a framework class, the module was compiled and size measured. This was done several times, deriving from a different class each cycle.
3. A second new class was added, the first one was changed back to not deriving from any class. The module was compiled and size measured for reference (**).
4. Both of the test classes where derived from a framework class. This was done several times, deriving from different classes each cycle.

The test results are shown in Table 9. The effect of deriving from framework classes is easily visible. With just two classes derived (from `CAknAppui` and `CAknForm`), the resulting module size increased 121% from the original implementation, for example.

Class 1 derivation	Class 2 derivation	Module size	Increase (bytes)	Increase (percentage)
-	n/a	4 132*	-	-
CAknAppUi	n/a	6 664	2 532	61 %
CAknForm	n/a	7 348	3 216	78 %
-	-	4 400**	-	-
CAknForm	-	7 444	3 044	69 %
CAknForm	CAknForm	8 196	3 796	86 %

Class 1 derivation	Class 2 derivation	Module size	Increase (bytes)	Increase (percentage)
CAknAppUi	-	6 768	2 368	54 %
CAknAppUi	CAknForm	9 712	5 312	121 %
CAknView	-	5 796	1 396	32 %
CBase	-	4 452	52	1 %

Table 9. Effect of derivation on module size.

To get some insight into how the “derivation effect” would show in a complete application, the modules of the case study application were compared. The results are shown in Table 10. Although the “Lines of code”-to-“binary size” ratio is not the same from module to module (different lines produce different amounts of code), the difference should not be very large. The results shown in Table 10 indicate a notable impact of derivation overhead in the UI module.

	Engine	UI	Ratio
Lines of code	989	1 301	132 %
Binary size	4 132	13 100	317 %

Table 10. Comparison of UI and engine modules.

As a rule of thumb, UI modules of applications cannot practically be optimised beyond a point that is reached pretty quickly. For example, nearly half of a 13 kilobyte UI module is probably composed of binary bloat caused by derivation. If we wanted to cut 30% off the module size by just refactoring code (without touching the derivation), we would have to shrink our implementation by a whopping 60%. That is just not possible.

It is noted these are important findings and they should be studied more closely and accurately. However, such work is out of the thesis’ scope and is not discussed here further.

7 CONCLUSIONS

There are multiple, practical, approaches and methods for optimising software size. The methods are divided into two categories by implementation cost. Methods that involve modifying existing code and writing new code are hard and expensive to implement. Usage of these methods requires skilled software engineers and thorough testing. Methods that involve transforming existing data files into a more compact form by software are considered easy and cheap to implement. Even less experienced engineers can take advantage of these methods and with less testing effort. Best results are achieved by mixing methods from both categories with a carefully balanced blend.

Technical architecture specification is the single most important process stage affecting software size. A good specification does not guarantee an optimal implementation; constant monitoring and measurement are needed. However, mistakes made at the architecture level cannot be corrected in the implementation if the specification is being respected. Software size aspects must be handled and determined in architecture specification.

There are considerable overheads connected to derivation in C++. This shows up especially with complex frameworks that are composed mostly by derivation. A model example of this is, sadly, Symbian's UI framework. It uses derivation to full extent. The Java UI implementation /4/, on the other hand, is an example of opposite design. The class hierarchy is kept lower and components are customised by parameterisation and aggregation, not derivation. It is recommendable to initiate further research for finding alternatives to the current Symbian UI framework usage model.

Small memory software construction requires skilled engineers. It requires disciplined project managers to keep the engineers doing the right things at the right times. It requires thorough, regular, code reviews. Too few people are concerned about the size of their software. Attitudes must change.

8 REFERENCES

/1/ Symbian, Ltd. [Internet]. Location: http://www.symbian.com/ [Referenced 19.10.2001].

/2/ James Noble, Charles Weir. Small Memory Software. Addison Wesley Longman, Inc. 2000. ISBN 0-201-59607-5.

/3/ Psion, Plc. [Internet]. Location: http://www.psion.com/. [Referenced 19.10.2001].

/4/ Jonathan Allin. Wireless Java For Symbian Devices. Symbian Press. 2001. ISBN 0-471-48684-1.

/5/ ARM, Ltd. [Internet]. Location: http://www.arm.com/ [Referenced 19.10.2001].

/6/ Steve Furber, ARM System-on-Chip Architecture. Addison Wesley Longman, Inc. 2000. Second Edition. ISBN 0-201-67519-6.

/7/ Martin Fowler. Refactoring: Improving the Design of Existing Code. Addison Wesley Longman, Inc.1999. ISBN 0-201-48567-2.

/8/ Nokia 9210 Communicator [Internet]. Location: http://www.nokia.com/phones/9210/. [Referenced 24.10.2001].

/9/ MultiMediaCard Association [Internet]. Location: http://www.mmca.org/. [Referenced 03.11.2001].

/10/ Memory Stick [Internet]. Location: http://www.memorystick.org/. [Referenced 03.11.2001].

/11/ Martin Tasker et al. Professional Symbian Programming: Mobile Solutions on the EPOC platform. Wrox Press Ltd. First printing, 2000. ISBN 1-861003-03-X.

/12/ Bloat City [Internet]. Location: http://www.wec.ufl.edu/staff/hydew/comp/os/nt_bloat.html. [Referenced 22.10.2001].

/13/ Robert C. Martin. Design Principles and Design Patterns. Available: http://www.objectmentor.com/publications/articlesBySubject.html [Checked 04.11.2001]

/14/ Robert V. Binder. Testing Object-Oriented Systems: Models, Patterns and Tools. Addison Wesley Longman, Inc. 2000. ISBN 0-201-80938-9.

/15/ Digia EUnit. [Confidential]. Available: Digia Intranet.

/16/ Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object Oriented software. Addison Wesley Longman, Inc. 19th printing, January 2000. ISBN 0-201-63361-2.

/17/ David Salomon. Data Compression: The Compete Reference. Springer-Verlag New York, Inc. 2000. 2nd edition. ISBN 0-387-95045-1.

/18/ ZLib [Internet]. Location: http://www.gzip.org/zlib/. [Referenced 30.10.2001].

/19/ Kyle York. A Random Access Compressed File Layer. C/C++ Users Journal, July 2001. CMP Media LLC. ISSN 1075-2838.

/20/ Digia Software Process. [Confidential]. Available: Digia Intranet.

/21/ Rational Software Corporation. [Internet]. Location: http://www.rational.com/. [Referenced 20.11.2001].

/22/ Kimmo Hoikka. DSP Technical Specification Guideline. [Confidential]. Available: Digia Intranet.