Lappeenranta University of Technology

Department of Information Technology

Messaging Application Engine for Symbian Platform

Subject approved by the department council on 22nd January 2004

Supervisors:	Professor, D.Sc. Lampinen Jouni (LUT)
	M.Sc. Asikainen Pauli (Nokia Corporation)
Instructors:	Professor, D.Sc. Lampinen Jouni (LUT)
	M.Sc. Kallio Petteri (Nokia Corporation)

Author:	Eronen Mikko
Address:	Insinöörinkatu 48 B 55
	33720 Tampere, Finland
Mobile:	+358 40 5419 486

Date and signature:

____

ABSTRACT

Lappeenranta University of Technology

Department of Information Technology

Eronen Mikko

Messaging Application Engine for Symbian Platform

Master's Thesis

2004

71 pages, 33 figures, and 7 tables

Supervisors:	Professor, D.Sc. Lampinen Jouni
	M.Sc. Asikainen Pauli
Keywords:	application engine, messaging, Symbian, Unified Process, pattern

This thesis describes the development of a messaging application engine for the Symbian platform. The whole application was required to respond to missed calls with predefined short messages according to the rule set defined by the user. The non-functional requirements included mitigated resource usage and possibility of reuse. Thus, the objective of this work was to develop an engine that encapsulates the user interface independent and reusable functionality of the application.

The Unified Process - an iterative, use-case driven, and architecture centric software development process - guided the development work. It also encouraged other industry best practices, such as the use of patterns and visual modeling with the Unified Modeling Language. Patterns were utilized during the development and the software was visually modeled to facilitate and clarify the design. The existing services from the platform were harnessed to minimize the development time and the use of resources.

The main engine responsibilities were assigned to be the message sending and the storage and checking of the rules. Different areas of the application, namely the application server and the user interface, could use the engine and it had no dependencies to the user interface layer. Thus, the resource usage was decreased and the reusability was increased. The message sending was implemented with ordinary Symbian platform mechanisms. The rule storage was realized with the developed persistence framework that separates the internal and external rule formats. Relational database was selected as the external format in this case. The rule checking was carried out with conventional object interworking.

The main objective was reached. This and the other estimated good results of reusability and mitigated resource usage were figured to derive from the use of patterns and the Unified Process. As the project was a small-scale one, good down scalability of these methods was demonstrated. They were also noted to support and encourage simultaneous learning with the development, which was essential in this case.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto

Tietotekniikan osasto

Eronen Mikko

Viestintäsovelluksen ydin Symbian-alustalle

Diplomityö

2004

71 sivua, 33 kuvaa ja 7 taulukkoa

Tarkastajat: Professori, KTT Lampinen Jouni

FM Asikainen Pauli

Hakusanat: sovellusohjelma, viestintä, Symbian, Unified Process, suunnittelumalli

Tämä diplomityö kuvaa viestintäsovelluksen ytimen kehitystyön Symbian-alustalle. Koko sovelluksen vaatimuksena oli vastaamattomiin puheluihin vastaaminen ennalta määritellyillä tekstiviesteillä käyttäjän määrittelemien sääntöjen mukaisesti. Ei-toiminnallisia vaatimuksia olivat resurssien käytön vähentäminen ja uudelleenkäytön mahdollistaminen. Täten tämän työn tavoitteena oli kehittää ydin, joka kapseloi sovelluksen sellaisen toiminnallisuuden, joka on käyttöliittymästä riippumatonta ja uudelleenkäytettävää.

Kehitystyössä ohjasi Unified Process, joka on iteroiva, käyttötapauksien ohjaama ja arkkitehtuurikeskeinen ohjelmistoprosessi. Se kannusti käyttämään myös muita teollisuudenalan vakiintuneita menetelmiä, kuten suunnittelumalleja ja visuaalista mallintamista käyttäen Unified Modelling Languagea. Suunnittelumalleja käytettiin kehitystystyön aikana ja ohjelmisto mallinnettiin visuaalisesti suunnittelun edistämiseksi ja selkiyttämiseksi. Alustan palveluita käytettiin hyväksi kehitysajan ja resurssien käytön minimoimiseksi.

Ytimen päätehtäviksi määrättiin viestien lähettäminen sekä sääntöjen talletus ja tarkistaminen. Sovelluksen eri alueet, eli sovelluspalvelin ja käyttöliittymä, pystyivät käyttämään ydintä ja sillä ei ollut riippuvuuksia käyttöliittymätasolle. Täten resurssien käyttö väheni ja uudelleenkäytettävyys lisääntyi. Viestien lähettäminen toteutettiin Symbian-alustan menetelmin. Sääntöjen tallettamiseen tehtiin tallennuskehys, joka eristää sääntöjen sisäisen ja ulkoisen muodon. Tässä tapauksessa ulkoiseksi tallennustavaksi valittiin relaatiotietokanta. Sääntöjen tarkastaminen toteutettiin tavanomaisella olioiden yhteistoiminnalla.

Päätavoite saavutettiin. Tämä ja muut hyviksi arvioidut lopputulokset, kuten uudelleenkäytettävyys ja vähentynyt resurssien käyttö, arveltiin juontuvan suunnittelumallien ja Unified Processin käytöstä. Kyseiset menetelmät osoittivat mukautuvansa pieniinkin projekteihin. Menetelmien todettiin myös tukevan ja kannustavan kehitystyön aikaista oppimista, mikä oli välttämätöntä tässä tapauksessa.

TABLE OF CONTENTS

ABBREVIATIONS

ADC	Analog to Digital Converter
API	Application Programming Interface
ARM	Advanced RISC Machines
ASR	Architecturally Significant Requirement
CASE	Computer-Aided Software Engineering
CPU	Central Processing Unit
CVS	Concurrent Versions System
DBMS	Database Management System
DCD	Design Class Diagram
DLL	Dynamically Loaded Library
DMA	Direct Memory Access
DSP	Digital Signal Processor
FURPS+	Functional, Usability, Reliability, Performance, Supportability, and more
GCC	GNU C Compiler
GNU	GNU is Not Unix
GPIO	General Purpose Input and Output
GRASP	General Responsibility Assignment Pattern
GUI	Graphical User Interface
IDE	Integrated Development Environment
IPC	Inter-Process Communication
LCD	Liquid Crystal Display
MIPS	Millions Instructions Per Second
MMU	Memory Management Unit
MTM	Message Type Module
OMAP	Open Multimedia Application Platform
OOD	Object-Oriented Design
OS	Operating System
PC	Personal Computer
PCB	Printed Circuit Board
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
SAD	Software Architecture Description
SDK	Software Development Kit
SMS	Short Message Service
SoC	System on Chip
SQL	Structured Query Language
TLS	Thread Local Store
UART	Universal Asynchronous Receiver Transmitter
UI	User Interface
UML	Unified Modeling Language
UP	Unified Process
USB	Universal Serial Bus

1 INTRODUCTION

When comparing modern mobile platforms with the personal computers of the mid 1990's, there are few significant differences from the performance perspective. Processing speed is about the same; a high-resolution color display and high-fidelity sound is available. Maybe the first significant difference that is thought of is the restricted memory quantity, both for execution and storage alike. Major part of the other differences arises from this restriction, which is even more conspicuous when mobile platforms are compared with modern personal computers. Despite of this memory restriction, modern applications and end-user experiences are expected and frequently delivered. Thus, clearly, something is being done differently.

Code reuse is an especially attractive concept in the memory-restricted devices. Therefore its implementation has gained more focus and it is already a reality in which frameworks, components, and other techniques are effectively used. At the platform level it is quite safe to state that the mobile devices available today would not have the rich feature set we have used to without a good middleware that enables a multitude of features with minimal amount of new application code.

Efficient code reuse is feasible even at the application level with well-designed object-oriented programming. The decisions about it are made during the iterative design phases of the application architecture. For a part to be reusable it must be reused, and reuse can be implemented in many ways. At application level this means that the application itself uses these parts repeatedly or that the application is going to be implemented for different platforms, in which case some parts are usually platform specific, for example the user interface. The non-platform specific parts should be designed so that they can be reused.

The messaging application engine developed during this thesis work was targeted for Symbian platform. As a mobile device platform it usually has restricted resources and focus on maintaining effective middleware. The application architecture, which includes the engine, was designed with the aim to use the middleware wherever possible and to use the engine by different areas of the application, thus reusing and being reused. The engine was implemented as a dynamic link library, so it is loaded only when it is needed and only once. The engine uses only standard Symbian operating system services, but the application was designed for Nokia Series 90 platform. The application user interface uses some services specific to it, which by no means diminishes the reusability of the engine by other Symbian based platforms.

The Unified Process was chosen as the process for guiding the work for its public availability and wide utilization. Many of its suggested best practices, such as the use of design patterns and the Unified Modeling Language, were also adopted. The Unified Process also implies early and regressive testing of the developed software. Therefore, it is particularly suitable for application engine development, because the reusability is inversely comparable to the amount of faults in the software.

2 SYMBIAN PLATFORM

Software is always created to some context called the target platform. In the worst case, though some may argue the best, it consists of hardware only. The Symbian platform targeted during this thesis is very much in the opposite corner. There is a diverse and complex operating system, called Symbian OS, handling the hardware and offering plethora of services to software developers. Native programming for Symbian OS is done with C++, but the supported dialect and usage is somewhat different from the standard C++ and requires some extra attention. Programming is done with a PC workstation and requires a chain of tools developed specifically for this purpose.

The material for this chapter is mainly from two sources, the original Symbian OS bible [18] by Martin Tasker et al. and its modernized adaptation [9] by Richard Harrison et al.

2.1 Hardware

The hardware architecture of the current Symbian platform has its roots in the first 32-bit platform made by Psion, the company that also developed the predecessor of Symbian OS, EPOC. They compared several different designs and implementations from major chipmakers, but in the end settled on the ARM architecture. Both Acorn's licensing model, which enabled incorporation to arbitrary designs, and architecture's technical suitability to mobile devices, as in terms of MIPS/$ and MIPS/Watt, made the choice clear [18, p 16].

A mobile device generally benefits from tight integration as both volume, and power consumption are decreased. Therefore modern mobile devices utilize so-called system-on-chip, or SoC, designs, which contain the CPU core and vital peripherals for the target functionality. Symbian OS requirements for the CPU core are that it must have an integrated memory management unit, or MMU, and cache, be able to operate in various privileged access modes, and handle interrupts and exceptions [14]. Typical peripherals in a SoC are timers, direct memory access, or DMA, controller, universal asynchronous receiver transmitters, or UART, digital signal processor, or DSP, and buffers. To construct a mobile phone some additional peripherals are needed and they can be placed on the printed circuit board, or PCB, of the device. Figure 1 illustrates this logical layering of the mobile phone hardware, to which also many Symbian OS devices conforms. This enables easy porting of Symbian OS as the code for particular CPU core or SoC can be reused in many products. [14]

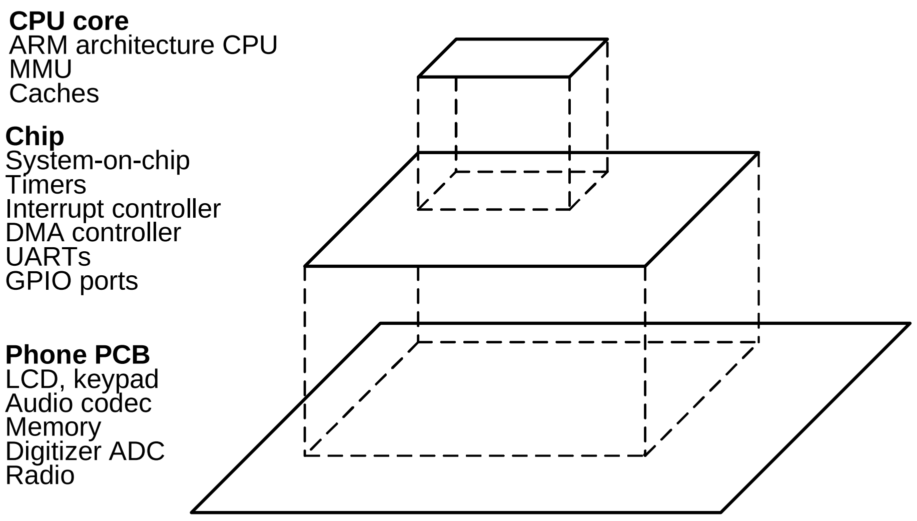


Figure 1: Logical layering of mobile phone hardware. [14]

Many SoC implementations are commercially available, but sometimes manufacturers use proprietary designs. One widely used implementation in Symbian platforms is Texas Instruments' OMAP1510, which has, among other things, built-in digital signal processor, or DSP, for accelerating the usage of multimedia formats, interface for camera and universal serial bus, or USB, and internal combined frame buffer and program/data memory [19].

2.2 Operating System

Symbian OS is intended to run on mobile communications centric devices such as mobile phones. This environment requires some operational characteristics from the operating system and Symbian OS addresses these specific needs starting with a suitable architecture and, for example, by providing a framework for handling low memory situations and a power management model.

The main architecture of Symbian OS is built from several layers; a common representation of this design is the layers architectural pattern described in [3]. One view of the layers in the OS is shown in Figure 2. Layered architecture has multiple benefits, especially in the restricted environments, in which the most obvious one is the resource usage effectiveness. By using the wide variety of the system software divided into different layers, the application designers can work only at the level of detail they need to, thus reusing code and saving implementation efforts. Layering also improves modularity and portability; therefore customizing the OS for different uses and devices is relatively easy.

The topmost layer shown in Figure 2 provides support for applications. Common application engines are one part of this support, including engines for address book, diary, and word processing applications. In this layer Symbian OS has also skeleton support for graphical user interface components, which are realized by the device manufacturers, thus unique look and feel can be obtained.

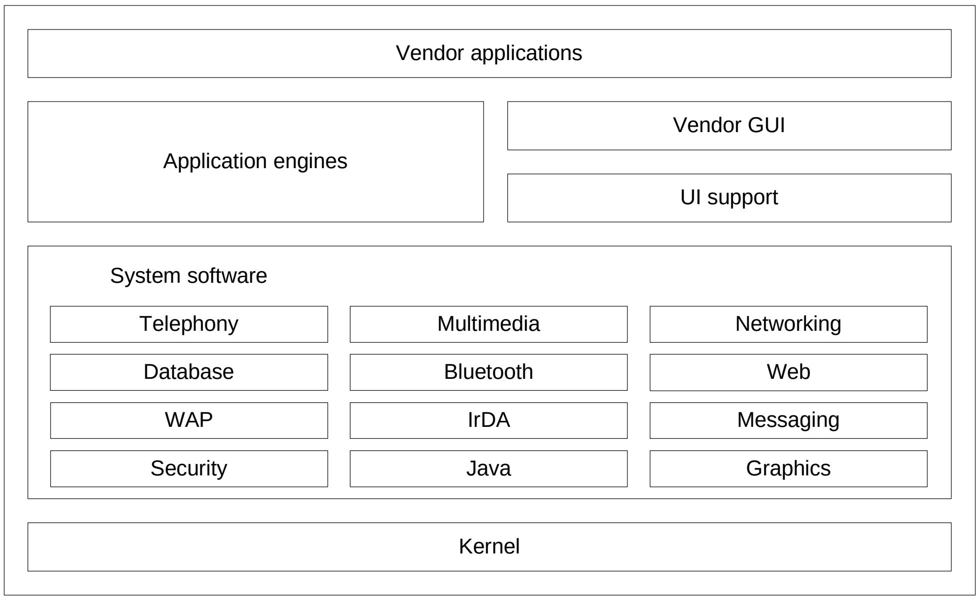


Figure 2: One view of the layers in Symbian OS. [14]

The kernel layer shown in the Figure 2 handles all hardware accesses with low level kernel library and device drivers, which are the only parts of the system that are truly device dependant. Kernel library includes support for all peripheral hardware that is essential to the operating system; device drivers add support and interfaces for other resources, for example different file systems. The kernel itself is based on the microkernel architecture, which follows the equivalent architectural pattern in [3]. Shortly put, microkernel separates minimal functional core – in this case approximately 200kB [14] - from extended functionalities and customer specific parts. From the application point of view this core functionality is offered by the user library, including support for inter-process calls, timers, memory handling, and event handling. Various system servers handle the major part of the other services. Because of the microkernel architecture and heavy use of servers, the kernel support for inter-process calls and client-server framework is optimized for low memory use and speed. The supported hardware access methods are kernel extensions and device drivers. Kernel extensions are usually made for peripherals associated with user input; device drivers encapsulate functionality that is offered to the upper layers [14]. Figure 3 shows kernel’s internal architecture.

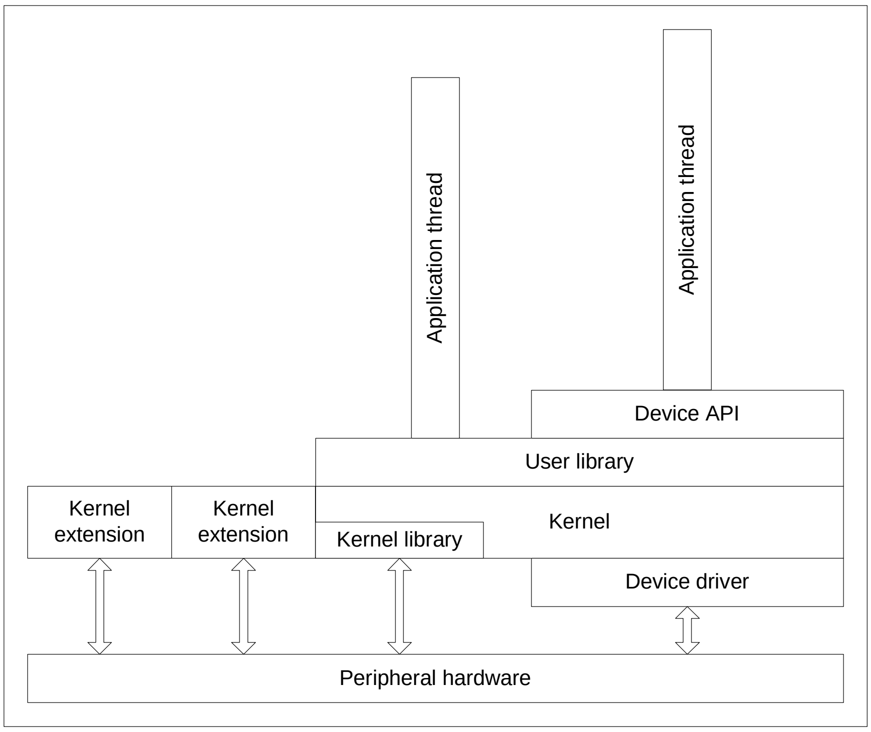


Figure 3: Symbian OS kernel architecture. [14]

Symbian OS has also an extensive collection of support for many industry standards; most of the covered areas can be seen in the system software layer of the Figure 2. These services and the system services are handled by Symbian OS's client-server framework, so it is under heavy use. Servers providing services are running in the user space without any special privileges and they also must go through the kernel layers. Reliance on servers in some very common tasks calls for high optimization, one option is to make an assumption that only one copy of the server is running at any time; thus it can have a unique data area in the system memory, which enables context switching without flushing cache. The file server is one example of such servers. Some co-working servers are actually ran as threads in the same process, again having less overhead from the context switches.

Most operating systems have pre-emptive multitasking with several processes and threads as a basis for their operation and services. Symbian OS differs fundamentally in this aspect, as it is deeply event-driven with focus on non-pre-emptive multitasking [9]. This is natural for GUI systems that have the user in control, but usually the event-based operation is limited to the layers handling the GUI. In Symbian OS every native application and server is a single event-handling thread, which uses *active objects* to handle events non-pre-emptively. Active objects are the realization of one part of the event-handling framework in Symbian OS. Basically, an active object signs up as a listener for a particular event and when the event occurs, the framework calls one particular method - *RunL* - of the active object. Very often a server signaling completion of a task generates the event. Evidently, the client-server framework and the event-handling framework are quite interwoven. For application programmers, Symbian OS's event-handling schema enables light-weight multitasking with relatively little effort, programmer must just keep in mind that the code must complete quickly, because the framework has no way of interrupting it.

Errors are inevitable, at least in the development phases of any non-trivial software. Errors related to the syntax of the used programming language are noticed in the compiling phase, but semantic errors have run-time effects, which must be noticed and handled then, leaving the actual error tracking to the developers. The most serious error in software for a mobile device, which is supposed to run continuously for long periods of time, is resource leaking. Usually this means memory leaking, which in practice means that the memory usage of a running program increases over time and eventually an out of memory situation is encountered. Symbian OS has support for detecting this kind of errors in the early stages of development, but since the resources are usually quite scarce, they tend to run out at some point even without any programming faults. The situation must be recoverable in a way that user data isn't lost, and Symbian OS has a robust error handling and cleanup framework for helping developers to do exactly this. GUI applications have some of it implemented by the environment, in engine development this must be handled by the created test harnesses.

In mobile devices it is important to use power efficiently. Battery life, weight, and size are factors, which directly affect the appearance of a mobile device, affecting user's general opinion about the mobility of the device. Therefore a power management model is provided by Symbian OS, and it covers the whole system stating some requirements for it beginning from the hardware layer and ending to the GUI layer. The kernel layer shuts down unused peripherals and even the CPU if every thread is in the suspended state. At the application layer the model usage is usually implicit, because event-based design supports it by nature. Awareness of the possible power offs at any time is still needed in some cases, for example when displaying some time-dependant information. At more general level, optimized and resource saving design preserves also power, so it is encouraged for this reason also.

2.3 Programming

The main programming languages supported by Symbian OS are C++ and Java. C++ is the language used for developing Symbian OS itself, thus optimized system development and application programming is naturally done with it, so it was quite obvious choice for the application engine programming during this thesis work. Java is increasingly popular language for its device-independent development possibilities, and its support in Symbian OS is getting better all the time, but it still has some drawbacks in the performance and capabilities areas.

Symbian OS has been developed with C++, but with a strong dialect, which promotes strict use of its own fundamental types, naming conventions, and use of templates. C++ doesn’t insist on the number of bits used to represent basic types, so new fundamental types are defined [9, p. 53]. These types obey the Symbian OS community-wide naming conventions, which also define the way of naming classes, data, functions, and macros. All these rules make the learning curve steeper, but the result is more readable and easier to follow code.

C++ template usage in Symbian OS is extensive; it is used for collection classes, fixed-length buffers, and utility functions [9, p. 67]. But, again, the usage differs from the standard C++ way by introducing a thin template idiom. Its main objective is to avoid code duplication and it achieves this by providing the functionality in a non-typed base class from which the templated classes are derived. The derived classes then act as type-safe wrappers around the base class; this is implemented with inline functions, thus no extra code is generated. The usage of this idiom is hidden from the programmers; thin-templated classes can be used in the same way as the normal C++ templates.

Symbian OS implements string handling with descriptors. C++ strings aren't used, because memory management is so important; with descriptors the awareness of the memory handling issues is conveyed to the programmer. In terms of ease of use and functionality, descriptors fall somewhere between C and C++ strings. The base class for all descriptors is *TDesC*, its name implying that it is a type (leading T) and that it is a constant (trailing C). TDesC includes functionality for handling constant data, thus this functionality is in every descriptor in Symbian OS. Descriptors are also used for handling binary data; in this case their 8-bit variations are used. Figure 4 illustrates the relationships of the descriptor classes.

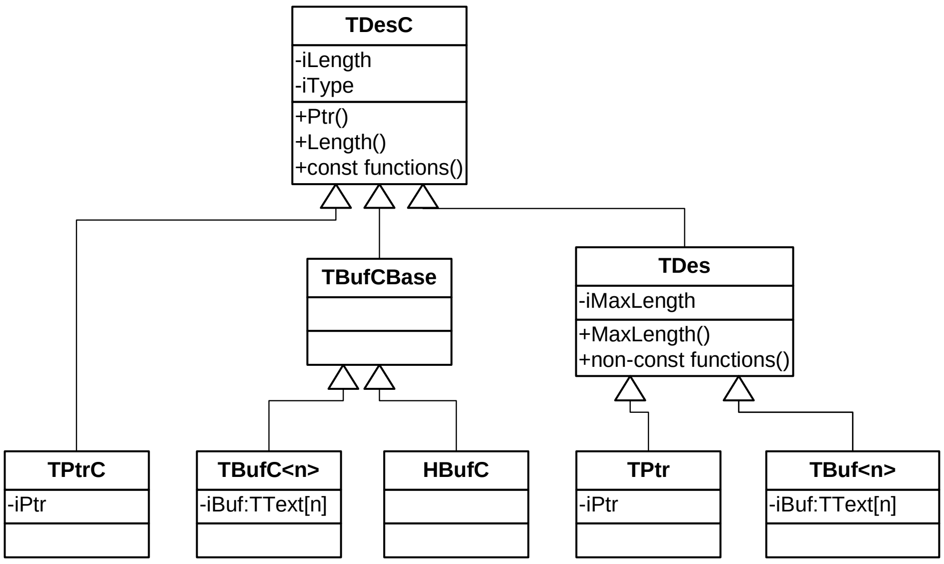


Figure 4: Descriptor classes. [18]

Error handling and the following cleanup should be in the programmer's mind at all times, especially when developing for Symbian OS. C++ exception handling isn't used, there is a proprietary mechanism via function *User:: Leave()* and macro *TRAP()*, which corresponds C++ *try-throw-catch* –triplet. Saying that a leave occurs or that a function leaves refers to this mechanism. All functions that may end up leaving must be marked with trailing L, this way other programmers can take it into account. The two-phase construction idiom is used for making sure that, the first phase, C++ constructor never leaves. The second phase construction, which is usually handled by *NewL* and *ConstructL* functions, is used for anything that might leave; during it the constructed class itself is inserted to the cleanup stack. The cleanup stack is the mechanism for cleaning up after a leave; objects on it will be deleted in that case. Therefore, all heap-based objects that are referenced by only automatic variables must be placed on the cleanup stack.

For more effective management of software development and deployment, the Symbian OS programs are grouped into different kind of packages. The package types are similar to the types that other systems use, namely .exe for programs with single entry point *E32Main()* and DLL for libraries of program code with potentially many entry points. Further, DLLs are divided into shared library DLLs and polymorphic DLLs. The difference is in how the interfaces are used. The former have a fixed API and have usually the extension .dll. The latter implement an abstract API and typically have an extension other than .dll, for example .app for applications for the Symbian OS application framework.

In Symbian OS, as in other C++ based object-oriented systems, APIs are mainly delivered as C++ classes. These classes are declared in header files, implemented in C++ source files, and delivered in DLLs [9, p. 64]. Loose classification of APIs divides them to library APIs and framework APIs, the former being the type that is called by your code to do something, the latter being the type that calls your code to do something. All native Symbian OS GUI applications are good examples of framework API usage; the application framework, which calls application's framework functions, controls them. Programmers just inherit the right framework interfaces for their classes and implement them. Application engines are usually implemented as library APIs, having a standard object-oriented design goal of hiding the implementation behind an interface. These interfaces must be exported from their DLLs for them to become visible to other software; corresponding definitions in the header files must define the same interfaces to be imported. Symbian OS has two special macros for this, *IMPORT_C* and *EXPORT_C*, which alter the actual implementation depending on the target platform.

DLL handling in Symbian OS is fundamentally different from the desktop world in one aspect; there can be no writable static data in DLLs – except one machine word per every thread, called thread-local storage, or TLS. The reason for this is the restricted available RAM; every DLL in every process would need a separate chunk of it, the minimal unit being 4 kB. This could quickly add up to unbearable - meaning several megabytes - amounts of wasted RAM space. In native Symbian OS programs this restriction is usually circumvented by design; the other solution is to use TLS as a pointer to a larger memory area, but this is comparatively slow. Bigger difficulties arise when porting software to Symbian OS, the easiest workaround is to capsulate the ported code into a server .exe which doesn't have this restriction. [9, p.38]

Symbian OS doesn't use all features of C++, actually some of them are listed [9, p. 71] as bad practices and their usage is discouraged. The default C++ inheritance, private, is seen as bad because it basically creates private ownership between the classes. Multiple inheritance is used just for interfaces; otherwise it is seen as more confusing than helpful. Overriding nontrivial virtual functions can be confusing, it is better to leave virtual functions empty or make them purely virtual. Just-in-case tactics, for example in making functions virtual or protected, are seen as clear evidence of unclear thinking and something to be avoided.

It may seem that programming for Symbian OS is quite restricted and requires a lot of time to become acquainted with, and actually both of these claims are quite right, but at the same time the overall benefits must be seen. The frameworks almost force creation of quality code, the conventions increase readability, and guided practices make things clearer; thus after a while it's easy to justify the trade.

2.4 Development Tools

Development of native Symbian OS software starts always, excluding the inner circles, with the same thing: getting an appropriate software development kit, or SDK. These are publicly available from Symbian Developer Network web pages [17], and can also be ordered for specific hardware platforms from several vendors. The SDK contains the most important development tools, including an emulator, the GCC compiler, and linker. Thus, anyone with a decent PC can start software development with little extra costs. In practice, an integrated development environment, or IDE, which makes it possible to edit, build, and debug software within the same GUI, is required for any non-trivial work. Microsoft Visual C++ used to be the de facto IDE for development, but recently MetroWerks CodeWarrior, which has, for example, enabled on-target debugging directly from the IDE, has taken the place.

The emulator is the most usual starting point for new development; most Symbian OS software is developed first on the emulator and only then on real target hardware. The emulator doesn’t emulate any specific hardware, but instead the Symbian OS environment. The emulator and all other software running on it are actually compiled as Windows executables, which causes problems sometimes. Mostly these are caused by the fact that the emulator is a single-process environment, unlike the target environments, which are multi-process. Usually this is transparent, as the application processes are emulated with threads, but some issues must be taken into account when servers or console applications are developed. These issues are handled by using compilation time flags, thus a little different programs are compiled for the emulator and hardware.

The emulator appearance is modified for different UI styles with an initialization file. It defines the size of the screen area, the used hardware buttons, and a bitmap showing the buttons. Figure 5 shows Nokia Series 90 platform emulator modified for Nokia 7700 media phone [13].

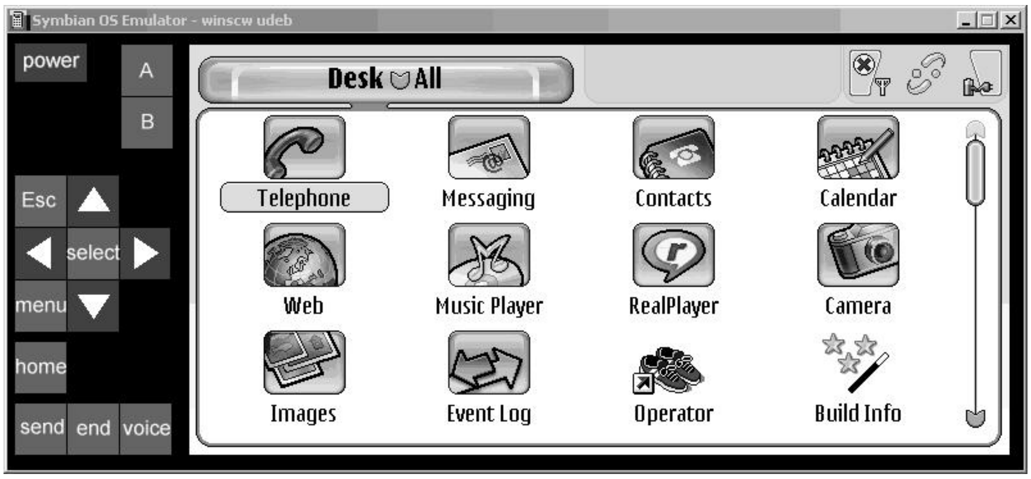


Figure 5: Nokia Series 90 platform emulator.

Obviously all development can't be done with the emulator, for example testing the messaging software developed during this thesis required real target hardware, which actually could send short messages. This implies additional development tool requirements, because the software must be, somehow, copied and executed on the hardware. The easiest way of doing this is to use the SDK tools to create a Symbian OS installation package, copy it to the supported external storage, and to install it on the hardware from there.

The emulator builds of the used Symbian platform are made with the compiler of the IDE. The builds for target hardware are made with a specially modified version of GNU C++ compiler, which is included in the SDKs. Another GNU tool, make, is used for middle-level control of the build process; high level control is handled by a set of proprietary command-line and Perl scripts, or the IDE. The developer has to provide a couple of information files - stating target platforms, used source files and libraries, and other project specific variables - for the scripts; after that the tools can generate the needed make-files and build the project.

3 UNIFIED PROCESS AND ITS UTILIZATION

Programming can be fun, challenging, and rewarding experience at many levels of complexity, if it is practiced in humanly comprehensible parts. Failing to do that leads quickly to frustration and confusion, and finally to a disappointment about the quality. Before the software industry even existed, the same problem – of handling complexity - was recognized in the other industries and it was confronted with established guidelines for dividing the work and guiding its progression – namely processes. A process gives the context for the work; this is especially important in the software industry, because the final products, as software, aren't too tangible. This was recognized early and several processes were created and used; the current crown jewel of this evolution maybe being the Unified Process for its public availability and wide utilization. For these reasons it was also chosen for the software development done during this thesis; by its nature it was adapted to the needs of this particular project, for example, by giving less focus to some parts that aren't so important in a one-man team.

This part was mainly guided by Craig Larman's great introductory writing about applying object-oriented methods [12], which uses the Unified Process as a sample process in its case study. The other general source was the original description of the Unified Process by Ivar Jacobson et al. [11].

3.1 Unified Process

The Unified Process, or UP, is a software development process - a set of activities needed for transforming user's requirements into a software system. In the core of the UP are three concepts, which make it unique; it is iterative and incremental, use-case driven, and architecture-centric. These all are commonly accepted best practices, which UP combine into a cohesive and well-documented description. In addition, an integral part of the UP is the use of the Unified Modeling Language [2], or UML, which is a tool for visually modeling software in a common way. The UP promotes also a long list of other best practices, but with adaptability and agility, thus no activities should be engaged without a real use for them.

Iterative development means, in the UP, that the main project is divided into several consecutive mini-projects called iterations. Each of these fixed length, for example four week long, iterations produce a tested, integrated, and executable system; maybe excluding the very first iteration. The iterations include requirements, design, implementation, and testing activities of their own. The result is never a throw-away prototype, but a production-grade subset of the final system; the difference between the results of two consecutive iterations is therefore called an increment. Further, iterations are organized across four major phases: Inception, Elaboration, Construction, and Transition. Table 1 shows roughly the contents of these phases.

Table 1: Phases in the UP. [12]

Phase	**Contents**
1. **Inception**	Approximate vision, business case, scope, vague estimates.
2. **Elaboration**	Refined vision, iteration implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates.
3. **Construction**	Iterative implementation of the remaining lower risk and easier elements, and preparation for deployment.
4. **Transition**	Beta tests, deployment.

The incremented result, and the end of an iteration, is a minor release. These releases can be used for getting feedback from all of the connected parties, for example users, developers, and tests (such as usability). This continuous feedback makes sure that it is possible to adapt early to changing requirements. These changes aren't supposed to be fundamental or chaotic, but more like a force closing the inevitable chasm between the initial requirements and their interpretation, or users and developers. Even though relatively major changes would be needed, it is better to detect and handle them early – and iterative development provides a mechanism for this.

An iteration endpoint, when some more significant - and formal - evaluation and decision occur, is called a milestone. Milestones are usually between the phases, for example after the Inception, when it usually must be decided whether to continue the project or not. The last iteration in the project's development cycle, and the Transition phase, ends with the final release. Figure 6 illustrates terms associated with the schedule in the UP.

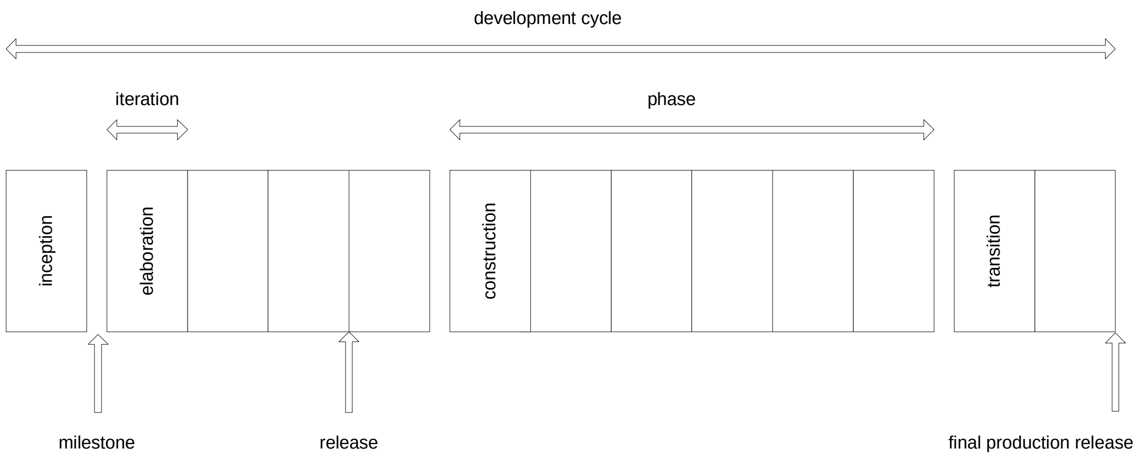


Figure 6: Schedule-related terms in the UP**.**

At a quick glance the phases of the UP can be mixed up with the phases of the old waterfall model, where certain activities are concentrated on a certain phase. The UP phases include all kinds of activities, just the relative amount of work between them changes during a full development cycle. These activities are described within different disciplines, such as requirements, and the results of them are called artifacts. In the UP an artifact means a general product of work and can be, for example, code, text document, or model. The models are abstractions of the system, describing it from a certain viewpoint and at a certain level of abstraction. The UP provides a carefully selected set of models, which tries to give every participating party a suitable viewpoint to the system. These models act also as the top-level containers for most artifacts; for example, a file containing a description of a class belongs to the Implementation Model, which is an abstraction of the system from the programmers' point of view. Different models are products of the corresponding disciplines during the project; Figure 7 illustrates the distribution of work into disciplines during the iterations, and lists the corresponding models.

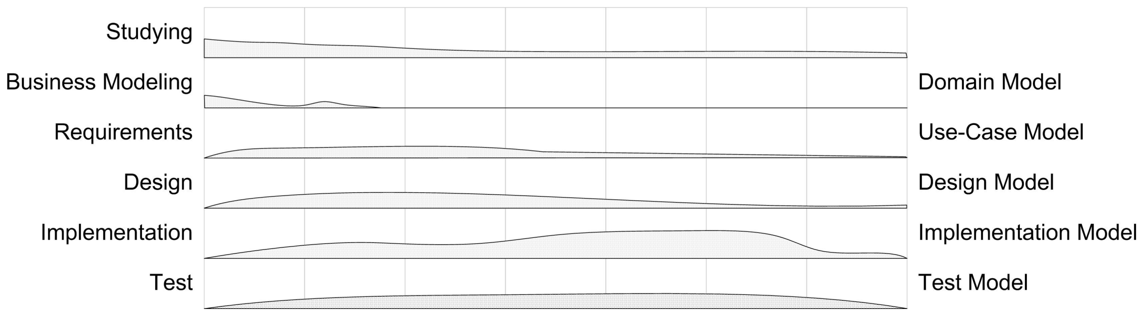


Figure 7: The disciplines that had focus during this thesis work, approximated work division between them, and the corresponding UP models.

Requirements are capabilities and conditions to which the system – and more broadly the project – must conform [11]. The UP encourages use-case driven development; one corollary is that requirements are primarily recorded with use cases. A use case is a described set of functionality in the system that gives an actor, e.g. a user or other system, a result of value through interaction; i.e. gives support in achieving some goal. The initial high-level system requirements are listed in the Vision artifact, which are then elaborated into use cases and stored in the Use-Case Model.

The UP categorizes requirements according to the FURPS+ (Functional, Usability, Reliability, Performance, Supportability, and more) model [8], from which the use-cases cover the functional category, but may also include the related non-functional requirements. UML diagrams may be used for illustrating the names of use cases, actors, and relationships, but they are secondary in the use-case work. The most important artifacts are the written use cases, which are stories about using a system; thus creating use-cases is mainly a writing activity. Because finally all success and alternative scenarios are included into a use case, it actually describes a set of functionality. However, use-case driven development doesn’t mean only specifying requirements, but also using use cases for driving development, implementation, and testing; the influence flows through all disciplines via their corresponding models. This way the attention stays on adding value; unnecessary functionality never realizes. Table 2 shows an example of a briefly written use case.

Table 2: Handle call use-case written in brief format.

A man calls with his mobile phone to the user who is not available at the moment. The user has anticipated this and wants a message to be sent to the man in this situation. Thus, the user has set a condition rule to match this situation. The user's mobile phone detects this condition, informs the user about it, and when the call is missed, a previously defined message is sent to the man.

The architecture of a system is driven by both functional and non-functional requirements. In the process of forming the initial architecture, first the most important factors must be identified, described, and prioritized; these should be documented in a factor table, which is a part of the Supplementary Specification artifact. Most of these factors can be found from the FURPS+ categories; in the UP these factors are called architecturally significant requirements, or ASRs. Use cases describe the functional requirements and give the context for the non-functional ones, thus some of the most important use cases must be written in parallel with the initial architectural analysis. Architectural analysis produces also solution alternatives for the ASRs, which leads into architectural decisions.

In the UP, the artifact used for storing all architectural information is the Software Architecture Description, or SAD. Just as the different models describe the system from different viewpoints, the SAD describes the architecture of the system with analogous views. The UP suggests six views of the architecture; the main difference between these and the models is that the SAD describes only those things that have some architectural significance, both of them use the UML and text as vehicles. Also the identified alternatives and the motivation behind the selected solution are described. Thus, the SAD functions as the key document for explaining the major ideas to different stakeholders; it can be used as an introductory document and as a reference when architectural changes are considered. Table 3 lists the suggested views and their function.

Table 3: Architectural views in UP. [12]

View	Function
Logical	Static view of the subsystems and other major elements.
Process	Processes and threads; generally a run-time view of the system.
Deployment	Physical deployment of the system.
Data	Persistent data related issues; relational databases and such.
Use case	Architecturally significant use cases and other requirements.
Implementation	Summary of the noteworthy implementation issues.

The Unified Modeling Language has been an integral part of the UP from the beginning, it contributes to the UP a standard way to visualize, specify, construct, document, and communicate the artifacts of the software system being built. Still, the UML is only a modeling language, it is not tied to a specific process, and therefore a wider variety of processes could have adopted it. The essential theme with the use of the UML is visual modeling, which is natural for humans; it is convenient to have a common language for communicating in this natural way.

As an agile process, the UP gives lots of freedom to decide which of its suggested best practices to use, as long as the core principles aren't forgotten. The project that was conducted during this thesis work was a light one; a one-man team with a goal to produce a relatively simple application for a relatively challenging platform. Thus, the agility of the UP was in test, could it really bring some value to a project of this size? The following chapters first roughly describe its utilization throughout the project and after that the final results are shown in more detail; let's see what the answer is.

3.2 Inception Phase

The Inception phase of the UP is about deciding the project's high-level goals, feasibility, and if it is worth further investigation. In this case the project was learning oriented and was going to proceed anyway, only the goals had to be selected so that they would support learning from the target area, namely messaging in Symbian OS. Ideas for such a project were gathered and one particularly interesting and suitable was chosen. The idea was for an application that replies missed calls with SMS messages according to certain rules. Among engineers, it seems to be a standard that a neat name for a piece of software must be invented immediately after its initial recognition; consequently, the software was named as ReplyMate.

ReplyMate is the name for the full application, including the user interface for the Series 90 platform, or more particularly for Nokia 7700 media phone [13]. At the time of the Inception phase the Series 90 platform wasn't revealed yet, and everything related to it was regarded as confidential. As one goal of the project was to produce suitable material for this thesis work, which had to be public because of the corporation guidelines, it was seen best to use the presumable logical division of the software as a basis and use only public material for some parts. This same constraint led to use of the UP also, thus at the time of the Inception phase it really wasn't identified as that, but its goals were well served by identifying these issues and adopting accordingly.

After the initial idea of the software was accepted, a more experienced developer quickly drafted a requirements document to guide the development work. That document was handled as a basis for the Vision artifact in the UP; it stated the high-level goals and some constraints of the software. These included also some quite detailed user-interface specific requirements, which are natural when there is an ongoing effort for producing cohesively functioning software, but in the UP belong into the Supplementary Specification. In this case these two documents were combined, this was convenient as the result was still manageable and unnecessary overhead could be avoided. Other started artifacts were the Use-Case Model and the Iteration plan; Risk List and other more project-management related artifacts were deliberately neglected, as their communicational value was not needed and they would anyway have included mostly some obvious issues, such as a risk of not having enough experience in Symbian OS software development.

The UP suggests that most use cases are identified during the Inception, summarized briefly, and only few critical ones written in detail. This suggestion was followed and a few days were used for drawing use-case diagrams and writing the descriptions. It was recognized that the use-case work took more time than estimated, even though it was carried through briefly; this was clearly due to the ongoing studying process of the method and tools, such as Rational Rose [10]. To convey this in the documentation, one discipline was added for studying; it worked as an umbrella for all related activities: reading, writing thesis, coffee drinking with colleagues etc. Figure 8 shows the main use-case diagram which doesn't name the actor for Start up because it was not yet know at that time; Table 3 already listed one briefly written part of it.

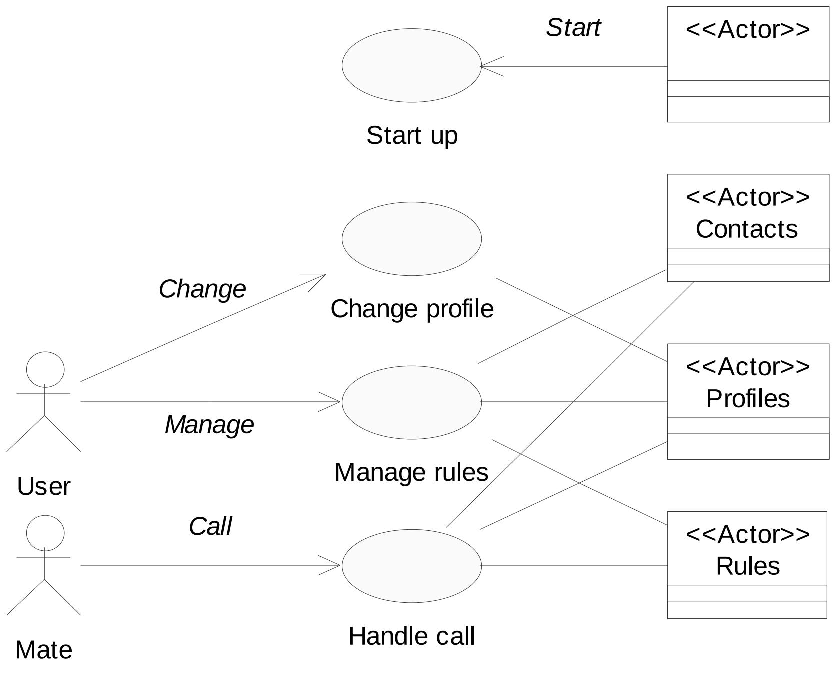


Figure 8: Main use-case diagram.

At some point of the use-case writing and studying of the Symbian OS, the initial architecture of ReplyMate started to take form. The user interface layer was going to be formed with the Symbian OS application framework; the other parts would have more choices, but it was evident that some kind of a data model, or engine, would be needed and something that could be always running in the background, even if the application user interface wasn't started. This led to the subsystem division presented in Figure 9.

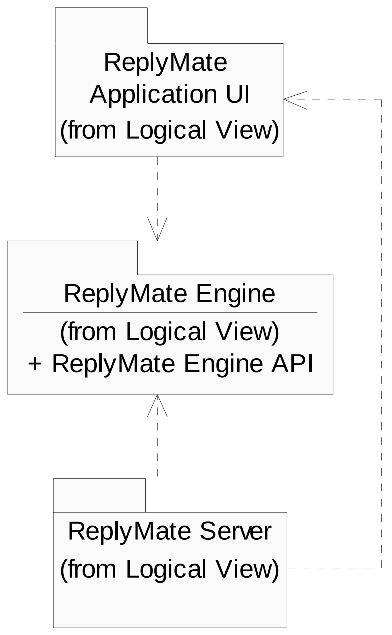


Figure 9: Logical view of the Inception phase architecture of ReplyMate application.

After the selected use cases were written and initial architecture formed, it was time to think forward a bit and create the Iteration Plan for the next iteration. The Handle call use-case was evaluated as the most important one, thus it should be approached first. Four-week long iterations felt right for this project, because it involved so much studying. Three Elaboration iterations, one counted per subsystem, would probably be enough to tackle the critical issues in those. As the project was going to be handled at the same time with some other duties, the time usage was estimated in full workweeks, not calendar weeks. This probably would generate some overhead and somewhat undermine the idea of compact development bursts, but it was estimated that this approach would not create big additional risks. With these estimations the project moved on to the next phase.

3.3 Elaboration Phase

After the brief examination of the area in the Inception phase it was time to go further. The UP recommends a wide and shallow approach to the Elaboration phase, meaning that every part of the system is examined, but still leaving the main implementation efforts to the Construction phase. The main goal is to have a robust architecture and interfaces on which to build, which makes it possible to concentrate on smaller parts at a time – or divide the work. The riskiest and most important issues should be recognized and tackled by implementing them at least partly, and releasing the working and tested increments after every iteration. In this case some discipline was needed, as the rush to code was already producing some side-projects, which all weren't too useful.

3.3.1 Iteration 1

The Use-case model was already started in the Inception phase and it contained enough material for the first Elaboration iteration, so there was no need for writing more use cases in detail immediately. Instead, one suggested addition to the UP toolset by Craig Larman in [12], namely System Sequence Diagrams, or SSDs, was given a try. SSD is a visualization of a use case, focusing on illustrating and isolating the operations that an external actor requests from the system, which is handled as a black box. This was stated to be an important part of understanding the system behavior, thus some SSDs were created for the main use cases, but their usefulness was questioned in this case. SSDs are probably a good way to communicate the materialization of the use cases, so there was no real need for them in this project. Consequently, also another related UP tool for describing system wide state changes, contracts, was discovered to be unnecessary. Figure 10 shows one SSD.

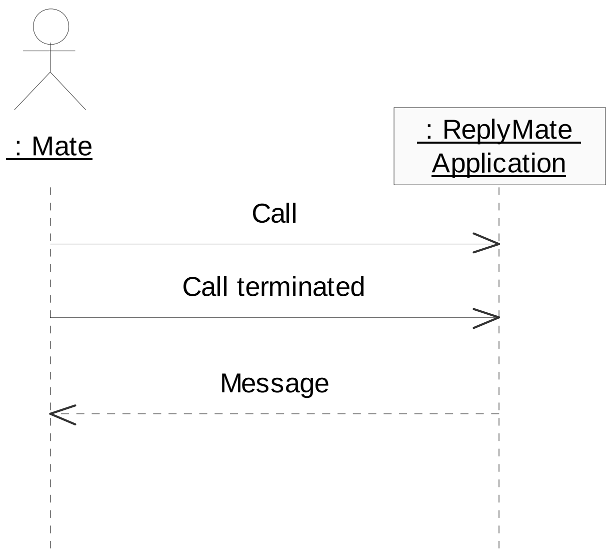


Figure 10: SSD for Handle call use-case.

The beginning of an Elaboration iteration includes, quick - in experienced hands, analysis of the problem domain. Analysis means finding the right things to do, after it is clarified, the probability that the system fulfills its goals is increased and the amount of just-in-case programming minimized. Creation of the Use-case Model is one part of this analysis in the UP; the other, and the most important part in any object-oriented analysis method, is domain modeling [12]. A domain model is a visualization of the real-world conceptual classes in the problem domain. In addition to the conceptual classes, it may also show their associations and attributes. The UML class diagrams are used for domain modeling in the UP; the Domain Model is the container for these visualizations. During this project, the noun phrase method was used for identifying conceptual classes. The Use-case Model works as the source in this method; the nouns and noun phrases in the use cases are considered as candidate conceptual classes. Some of the conceptual classes found with this method were quite abstract, but this was learned to be normal in a domain such as telecommunications. A domain model doesn't illustrate only the conceptual classes, but their associations and attributes also. Some effort was used for finding those, enough to identify the most important ones. Categorization lists and data type identification methods were found to be useful during this. Figure 11 shows the created domain model.

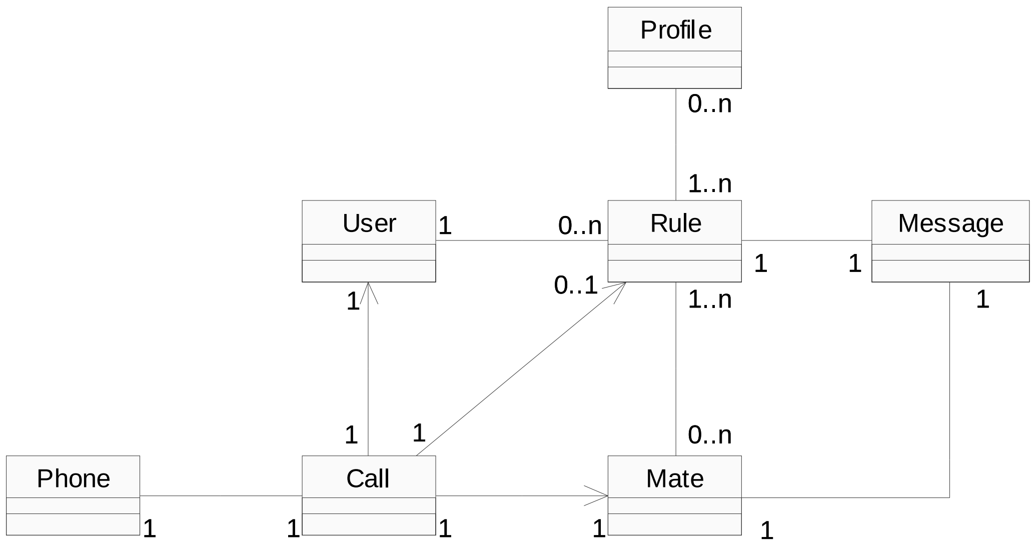


Figure 11: The domain model created during the first Elaboration iteration.

At this point approximately one week of the first Elaboration iteration was spent for investigating somewhat abstract issues. According literature, only a day or two should be used for the analysis in the elaboration iterations, but again the needed absorption of these new methods consumed some time. Now it was time to descend one abstraction level closer to implementation and start designing software objects.

Object design is very much about assigning responsibilities, which are basically of two types: knowing and doing. At the design phase the responsibility choices are usually considered in the process of creating interaction diagrams; the UML has two diagram types for them: sequence and collaboration. Sequence diagrams illustrate the flow of messages in a sequence from top to down and collaboration diagrams in a graph or network format. Many CASE tools can convert one diagram type to the other; thus, there normally is no need for drawing both, the more convenient one can be used.

The set of the illustrated interactions is not arbitrary; instead, the Use-case Model drives this activity. Thus, the use cases are realized with interaction diagrams. Several principles and patterns have been defined for guiding this work. The UP leaves the choice of methods open in this case, so some generally useful patterns for responsibility assignment presented in [12] were used. These patterns have been named as general responsibility assignment software patterns, or GRASP. The abbreviation is usually accompanied with word “patterns” for convenience. As usual with patterns, the ideas that GRASP patterns convey sound logical and self-evident. The value is more in creating a common language for communicating the decisions. The GRASP patterns were selected as part of this function in this thesis, thus some knowledge of them is needed later on. GRASP patterns and their solutions are listed in Table 4.

Table 4: General Responsibility Assignment Software Patterns, or GRASPs. [12]

Pattern	Solution
Information Expert	Assign a responsibility to the information expert – the class that has the information necessary to fulfill the responsibility.
Creator	Assign the responsibility of creating an object to a class that has a close relation to it.
High Cohesion	Assign a responsibility so that cohesion remains high.
Low Coupling	Assing a responsibility so that the coupling remains low.
Controller	Assign the responsibility for receiving or handling system event messages to a class that represents the overall system or a use case scenario.
Polymorphism	Assign responsibilities for the behavior to the types for which the behavior varies.
Pure Fabrication	To support better system design, assign a highly cohesive set of responsibilities to a class that does not represent a problem domain concept.
Indirection	Assign the responsibility to an intermediate object to mediate between other components or services.
Protected Variations	Identify points of predicted variation or instability and create a stable interface around them.

In parallel with creation of the first interaction diagrams, the design classes for the system started taking form. The conceptual classes defined earlier, which were already quite abstract, inspired their naming, thus the mapping was almost one-to-one at this point. Still, the difference is that the design classes model real software classes, thus they have also methods for implementing the messages illustrated in the interaction diagrams. In the UP the design classes are illustrated with design class diagrams, or DCDs, which are UML class diagrams. Before DCDs can be fully used, one aspect has to be considered, namely the visibility of objects. An object must have visibility to the objects it wants to send messages to. There are four common kinds of visibility: attribute, parameter, local, and global. Interaction diagrams define the requirements for visibility and DCDs illustrate it. Attribute visibility is illustrated with associations, other visibility types with dependencies. Figure 12 shows the first DCD of the ReplyMate server that handles the Handle call use case.

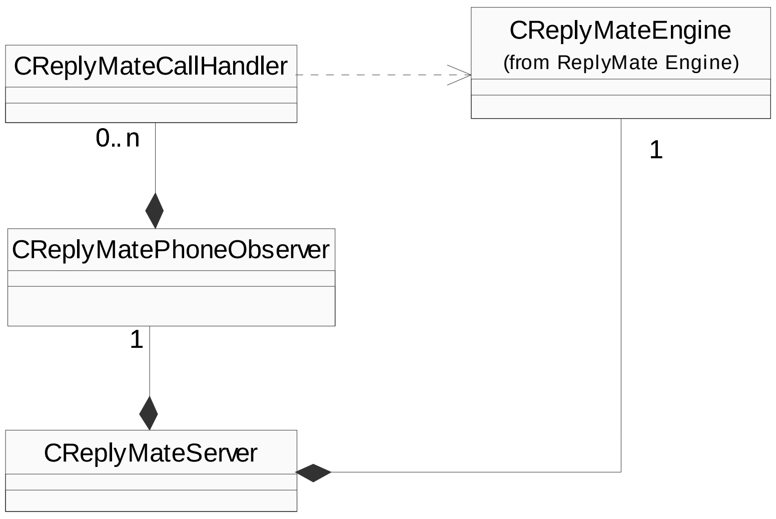


Figure 12: Initial design class diagram of ReplyMate server.

Design patterns, popular description of which can be found from [6], are applied during the object design phase for solving some distinct design problem; in this case most of the problems were yet to be discovered. Therefore, the design was guided more by plain gut feeling; design patterns would be applied when the problem domain was understood better.

It’s hard to imagine a software developer who wouldn’t have wished a few times that some more thinking had been involved before starting to code. After the described analysis and design, there was strong evidence that the thinking-before-doing part was fulfilled for this iteration. Approximately two weeks of the iteration was used, maybe one full week could be used for implementation, leaving plenty of time for the expected unexpected issues.

The handle call use case stated that implementation of the server part should be started first. It was called server even though no client-server functionality was planned, the name just implied that the component would serve some need in the whole application. The first message in its sequence diagram, *callMissed*, was coming from the system. To receive system events for missed calls, they probably must be requested; after studying the issue a little further, it turned out that specifically that kind of events weren't offered by the telephone subsystem. Instead, all state-changing events for the phone line should be requested and then deduce the call-missed state from those with a state machine. The identified line states and transitions are illustrated in Figure 13.

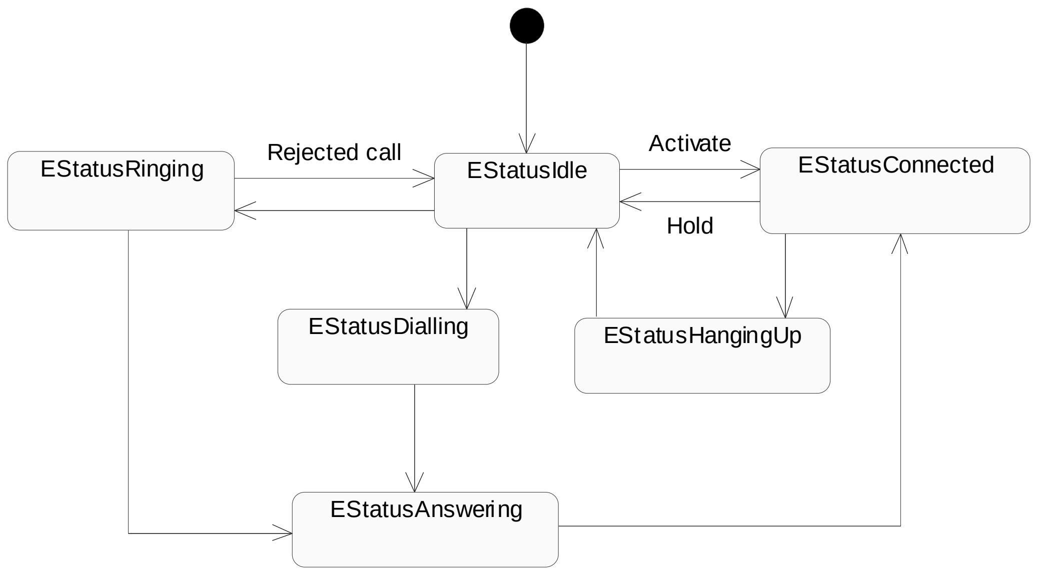


Figure 13: State diagram of possible line states.

While identifying the related states, one unfortunate feature of the system was discovered; it was impossible to deduce the call-missed case from only the phone line states. If the user decided to reject a call, the same transition would occur as if the call was missed, i.e. the calling party terminated the call. Therefore, some more information was needed for the server to detect missed calls, but the source for it was unknown. Because of this, also other possible mechanisms for the same functionality were considered, but as it seemed obvious that the missing information had to be available somewhere, the state machine version was still chosen and its implementation continued. It was implemented with a Symbian OS idiom of a *switch-case* structure in an active object, which would send a message to the engine after detecting a missed call.

An early vision was that the engine would have the “brains” of the application. It would have the functionality for storing and using the user’s rules for replying with messages. At this point this wasn’t yet required, so just an interface for using the engine was sketched. Instead of being just a stub, the Symbian OS message sending mechanism was studied a bit and a method that sends SMS messages to a parameter number was implemented. It could be used later on wherever the responsibility would be assigned to. The engine was a critical part of the system, so some test cases were written for it already at this point, also because of learning how to unit test in Symbian OS. A console test application was created for these purposes; it constructed the engine, called the stub interfaces, and evaluated the return values against the predefined ones.

Implementing these components took almost two weeks, thus there was only a couple of days left for wrapping up the first Elaboration iteration. First some time was used for updating architecture documentation and planning the next iteration. Deployment view was elaborated in the SAD, now it also showed the division of the application to different Symbian OS packages: application UI .app, server .exe, and engine DLL. The next iteration was planned to include - in addition to the normal UP practices - some application UI work to do something a bit more concrete, development of the server to detect missed calls and profile changes, and evolvement of the engine into a more useful state. The leftover time was used for studying Symbian OS details related to the upcoming issues.

3.3.2 Iteration 2

The second Elaboration iteration began with the same activities as the first one: analysis and design. This time these activities were finished much faster, a few days worth of writing use cases, updating design diagrams to reflect the changes that were made in last iteration, and designing further.

While playing around with the first server release, one new requirement came up; the user should be noted about the possible forthcoming action when the phone rings and it should be possible to cancel it. This also made it clear that the line states had to be observed; just using ready-made missed call information from e.g. the system log was not enough. In addition, this new UI functionality shouldn’t inhibit in any way the normal call handling process. The use case was updated accordingly, but the design was all but clear. At this point the Risk list artifact creation started to be reasonable, thus this issue was listed as a high-risk item in it, also some lower risk issues mentioned earlier were listed. The suggested mitigation action for the risk was to study the Symbian OS user interface programming by implementing a user interface that could be used for the realization of the Manage rules use case later on. As a shortcut, the Vision artifact stated that the UI of an existing application should be used as a model for the layout, thus all the efforts could be used on the implementation.

After some studying and a training course, it was noted that the Symbian OS application UI framework basically forces use of the model-view-controller architectural pattern [3] and many other issues. Therefore, the UI programming for the rule editor part was estimated to be quite mechanical and low-risk work, thus after implementing a small graphical application with some command buttons and menus, it was left waiting for the Construction phase. At the same time it was noted that the user notification requirement couldn’t be resolved with the same user interface, because – firstly - it would conflict with the normal telephone application and – secondly - in that case the UI should be loaded at all times, which would waste resources. Therefore, something else was needed. The framework for the solution already existed in Symbian OS and it answered exactly this need, namely servers needing a simple UI. It is called as Extended Notifier

Framework, which defines an interface for plug-in DLLs that are loaded at the boot time and can be invoked from non-UI programs via system calls. Still this idea had to be verified and one week was allocated for it.

Immediately after starting to study the notifier implementation it began to seem more challenging than expected. One reason for this was the utmost quality requirements; the notifier DLLs are loaded and ran in the same thread with the Uikon server, which is a system server that serves a large part of the UI API calls in the system [16]. Therefore, if a leave occurred in a notifier, it would result in a device reset. Further, as the notifier was going to be loaded at the device boot, its construction phase was critical. During the initial implementation, one malfunctioning notifier was installed to a testing device, which ended up in a boot loop that could be ended only by formatting the internal mass storage of the device. An additional problem was that the telephone application insisted on keeping the focus of control, thus an initial dialog-based implementation idea had to be dropped. The only option in sight was to implement a status pane - an analogous concept with Windows® system tray - control that would have identifiable graphics and report user's clicks to the server asynchronously. A part of the corresponding message could also be shown periodically with info messages, which stay on the screen for a while and don't block the telephone UI. After these considerations, the notifier was identified as a subsystem of its own and that in practice handling the increased complexity would require an additional Elaboration iteration, so the Iteration plan was updated accordingly.

The solution for detecting missed calls still awaited disclosure. No additional couplings to other system components were wanted, but after discovering that more detailed call event reporting for mobile calls was not supported even though hints of its existence were found, further efforts for more elegant approach were abandoned and quick and dirty method chosen: after a call would be rejected, its missed-call state would be checked from the system log of the device by using Symbian OS log engine functionality.

The last piece of missing information for the Handle call use-case was the current profile. Profiles are sets of behavior for a device, mostly concentrating on different alert effects such as ringing tones. As Nokia proprietary implementation their direct usage from the engine had to be avoided; at the same time the most logical place for their use – by Low Coupling GRASP - was the engine. The former force was stronger in this case, so the logical coupling was made from the server and the UI, but the implementation was still delivered with the engine as it was needed from both components. The designed class would act as an observer and a local proxy for profile information and depending on the need it could be fetched for the server or delivered via callback to the UI. This functionality also covered the Change profile use case, so it didn’t require handling anymore.

The notifier issue had used more time than expected, so the planned engine evolution had to be pushed to the next iteration where it would have the main focus. As there still were a few days left and some previous database design experience existed, a relational database for storing rules was sketched. Also flat files were considered for this, but their usage would have required some more efforts in mapping between them and objects, thus available and robust Symbian OS DBMS engine was chosen. After that the remaining time was used for updating the relevant documentation and planning a bit further.

3.3.3 Iterations 3 and 4

The third Elaboration iteration started again with a few days worth of analysis and design, this time concentrating on getting as ready use cases as possible for driving the engine development. After this the use cases actually looked almost finalized; probably some issues would arise during the Construction phase and UI implementation, but nothing major was expected. The planned engine design and implementation advanced much more easily than with the previous subsystems, which probably was caused by the facts that there were less Symbian OS specific issues. The problem domain was also better understood now, thus design patterns could be applied effectively. The results of this are described in Chapter 5.

During the engine design process, an interesting approach to testing was discovered from the Extreme Programming methodology explained in [1], namely creating test cases before implementing the tested functionality. This sounded to be worth a try and was taken into use; after that the test cases handled the interfaces of the engine classes one step ahead of the implementation, at the same time climbing the class hierarchy from bottom to top. In short term this increased work amount, also because some trivial test cases were written in enthusiasm of using this new method, which lead to an engine release that still didn't really offer any functionality to the external interfaces. This was a slight deviation from the plan, but as the trust on the implemented parts was increased in the process, it wasn't seen as a problem. Still, there was one full Elaboration iteration left and the development moved to it via the already standard procedures.

The fourth, and the last, Elaboration iteration began with a quick glance at all the handled issues so far. The situation was rather good; the only use case that was not handled was the Startup, which was left waiting for this iteration because of an advice given in [12]. It stated that the startup should be handled last; only at that point the preconditions for the other use cases would be known and could be addressed. As there already existed one component that was automatically loaded during the device startup, the notifier plug-in, it was also possible to start the server at that time. The server connects to the required services during its construction, thus their existence already at Uikon server startup time was manually checked and no needs for changes were discovered. Actually it was discovered that the notifier loading mechanism made it possible not to load the notifier at all if the server startup didn't succeed.

At this time a Symbian OS client/server training course was attended to; as a side product the ReplyMate server was modified so that it could receive and handle simple synchronous requests by clients. The architectural change to use a client/server model didn't seem so big after all and it would have the benefit of keeping only one copy of the rules in memory. Still, the benefit didn't seem big enough to justify the change at this point of development, so the existing design was kept. The limited client/server functionality was still left in the server, it made possible to check the availability of the server from the UI and to shut it down cleanly. A client interface was created for accessing this functionality and packaged with the engine.

Now the issue of finalizing the engine was focused on for a couple of weeks. This work continued straightforwardly from where it was left in after last iteration. The engine was completed in this time, at least all current use cases were designed to be realizable with it, and even some integration with the UI could be started. The main view of the UI was going to be based on a list box control that was derived from the Symbian OS *CEikListBox* control, which uses model-view abstraction. The model class could also be derived from, thus its methods were re-written to use the engine. Now that the UI could show the rules from the engine, the use-case driven development of actions for these rules could continue in the Construction phase.

The last few days of the Elaboration phase were used for updating all relevant documents so that they would be in line with the implementation. The architecture of the system felt ready, as it should be in this phase; the biggest change to it during the Elaboration was the introduction of the notifier. Some architecturally significant requirements were mentioned in the Vision artifact and a few more surfaced during the Elaboration phase. These were listed in the architecture documentation and their effects were considered and used as a driving force behind the design decisions. This work created the architecture for the system, but it had to be documented also. The main tool for this is to show the most important ideas as architectural views to the UP models; thus the information is selectively collected from the already existing artifacts. The most suitable time for finalizing this work was at this point and the results can be found in Chapter 4. After that the Construction phase could be started.

3.4 Construction Phase

In the Unified Process the Construction phase is the main development phase during which the first operational release of the product is realized. The analysis and design activities in the former phases have shown what to do and how to do it; the critical parts have also been implemented and interfaces stabilized to make sure that the design works. It is still possible and encouraged to capture more details to use cases and make small changes to the design, if something valuable is discovered. Architectural baseline is established in Elaboration, so major changes during Construction aren’t an option.

Depending on the project the relative proportions of the phases vary, in this case the proportions seemed to end up being upside down when compared to the cases handled in the literature. The reason behind this was that the rather steep learning curve of Symbian OS application development created additional risks and forced to handle some issues already during Elaboration. Also the planned iteration amount and length were explained by the needed learning, and naturally this need diminished all the time. Handling these issues had created more than just an architectural baseline, some parts that would normally be implemented in the Construction were already up and running. Therefore, only two Construction iterations were planned, both of which still being four-week long would probably give just enough time to complete the work.

The biggest remaining issues were related to the UI implementation, which can be quite time-consuming if every detail is honed to perfection. Therefore, a less ambitious approach was accepted; it had to be possible to complete all use cases, but it didn’t have to be as pleasant as possible. After all, the project was learning-oriented, in real business environment the UI specification would be created by an expert of that area. With this use-case focus the most obvious target for development was the Manage rules use case, as it was the main part missing from operational functionality. The use case acted as a container for several smaller use cases such as Activate rule, Add rule, and Delete rule. These were ordered by their importance and possible dependencies – Delete isn’t possible without Add – and divided into two iterations, after that their realization started.

Due to the way the engine was designed it could be treated as a black box during the UI design. Thus, the interaction diagrams that were created for the use cases at this point didn't anymore handle the internal functionality of the engine. Instead, the focus was on translating the engine responses into human readable form and vice versa. This was needed because the engine mostly stored references to other data sources in the device; for example, no contact names were stored, just their identification numbers in the contact database. Therefore, the UI had to fetch the relevant information from these sources instead of the engine. Plainly from the UI perspective this was against the Low Coupling GRASP, but the couplings were needed somewhere and by High Cohesion the UI was more suitable place for making these conversions. Also some other considerations related to this issue, those are handled in Chapter 5.

Near the end of the first Construction iteration, when Add, Delete, and Activate use cases were realized, the business side of things interfered with the project. As the project was mainly learning oriented and its goals in that sense were already served to some extent, the resources were wanted elsewhere. There was still some interest to finalize the UI part before documenting the whole process into the form of this thesis, as without the ability to modify the rules the application was quite useless to anyone who couldn't edit the rules by directly modifying the database. As the new allocation of the resources was rather challenging, this goal was abandoned after a while, at least for now. Thus, the Chapter 6, which illustrates the application UI, is in part based on design rather than implementation.

In addition to the Construction phase, also the Transition phase was missed. As its role in the UP is to roll out the created system to the end users and to make the final adjustments according their feedback, its existence in this project was quite unlikely already in the beginning.

4 ARCHITECTURE DESCRIPTION

Software is created to serve some function, which in turn shows up as requirements for the developers of the software. In the UP, this functionality and corresponding requirements are mostly captured with use cases, thus the process is stated to be use-case driven. The UP is also stated to be architecture-centric, and it means that architecture is what gives the form for the software under development. This form, and software architecture generally, means the larger scale elements of a system and their interconnections. As the application engine described in this thesis is one larger scale element, or subsystem, in the ReplyMate application, the requirements for it are derived from the whole application. Therefore, the application as a whole must be described and architecture is the most suitable vehicle for it.

The required functionality must fit into form created by architecture, but it must also leave room for inevitable evolvement. In addition to functional, also non-functional requirements must be taken into account when the architectural decisions are made. In many cases, and also in the project described in this thesis, the non-functional requirements have a major effect on the final architecture. These identified factors are called in the UP as Architecturally Significant Requirements, or ASRs. The ASRs identified during this thesis work are explained in Chapter 4.1. Also other, platform wide, ASRs were considered during the work, but they are not commented here.

Software has some form whether it is documented or not, but only documentation realizes architecture [11]. Software Architecture Document, or SAD, is the UP artifact used for documenting architectural issues. SAD approaches this task with several views to the system, which are used to describe the most architecturally significant static and dynamic aspects from a given perspective. These aspects are made even more visible by leaving details aside. In addition to describing the final system, also motivation and possibly considered alternatives should be explained, as it increases understanding and potentially reduces re-invention. The views to the ReplyMate software, which were considered most relevant, are shown beginning from Chapter 4.2.

4.1 Architecturally Significant Requirements

In the UP the ASRs are categorized according the FURPS+ model, in the case of ReplyMate all these categories got a few entries, also the ‘+’-category. In the UP these entries are documented in a factor table, which is a part of the Supplementary Specification artifact. Further, the solutions for these factors, their motivation, and alternatives should be documented with some sort of technical memos. These memos are a part of the SAD, in which they belong to a suitable view per memo. Here the ASRs are described briefly in the FURPS+ order with their solutions.

All main use cases, namely Handle call, Change profile, and Manage rules, had impact on the architecture of the ReplyMate software. Handle call and Manage rules had separate concerns, so by High Cohesion they were separated into different functional units, i.e. server and application UI. Still, they shared a need for a common mechanism for accessing the current rule set. This functionality was separated into an engine component, which also accommodated Change profile functionality, as both server and application UI needed it. Handle call was also identified to need a mechanism to inform the user about the forthcoming message sending. This had to be implemented in a way that it wouldn’t affect the normal call handling process. These functionalities were achieved with a notifier plug-in, which had to be implemented as a separate functional unit because of the used framework.

The rules needed to be persistent, in practice meaning that they had to be stored to the mass storage of the device in some way. A low level approach of using flat files for this was considered at first. In this case all low-level issues, such as the form of data in the files and accessing it, would have needed consideration. As there is a ready-made DBMS engine in Symbian OS, it was more resource-effective and convenient to use it; previous SQL experience had also some influence. It would have been possible to include the DBMS functionality into the functional engine classes or to use a single hard-coded mapper class, but these solutions would have been strongly in conflict with Low Coupling and High Cohesion. Therefore, a more elegant solution of persistence framework was chosen; it was also far more useful for learning.

The usage of two engines, which access the same database, needed functionality for opening the database in read-write or read-only mode and a way of keeping the data consistent. The solution was to use the Observer [6] pattern, which was well supported by the Symbian OS DBMS engine. The ReplyMate engine is by default constructed in the read-only mode, in which it also starts observing changes to the used database. When changes occur, the rule set is reloaded in a delayed manner to minimize disk accesses. The read-write version of the engine acts as a publisher via changing the database. Other option would have been the usage of the client/server API to communicate the changed rules, but it was abandoned for schedule reasons.

Usability can only be measured by making usability tests, so in this case the approach for handling usability requirements was rather intuitive. It was estimated, that if the behavior of the given model application UI was imitated, the usability requirements for the UI would be fulfilled. The functionality of the server part raised the biggest usability question, which partly led into changing the architecture by adding the notifier component. The question was about sending messages without user’s knowledge and giving possibility to cancel the action. Because sending short message is a billable service in most cases, it should be avoided even if the action would otherwise be justified. These usability considerations led into adding new functional requirements: it should be possible to cancel the ongoing activity and if the current action was already executed during a defined time window, it should be cancelled automatically.

The server and notifier components had tight reliability requirements. Both of them should never fail in normal circumstances, i.e. if the device was in a working condition. The application UI was seen as a less critical part, the user always could start it again, so it got only medium reliability priority. During the development the notifier was prioritized even higher than the server for reliability, the reason being that a notifier failure always took the whole system with it. The requirement for total reliability of the server implicated that either the engine should also be in the same level or that the engine accesses would be protected. As the protection was quite straightforward to implement, it was decided to be done anyways. Still, the engine reliability had high priority and thorough testing was needed.

The performance requirements were implied by the application area and platform. Restricted memory quantity states that the run-time memory footprint should be minimized, even if it means slower execution speed. The chosen solution isn’t optimal in this sense; the motivation for it is explained later on. The best option would have been to implement the engine functionality in the server and only offer a client API DLL for using it. Still, the chosen implementation of using a single engine interface makes it possible to change the underlying implementation to client/server, if that is wanted.

The application area of detecting and reacting to phone calls requires at least semi real-time functionality. For this reason the call detection was separated into an active object which has minimal other duties. When a state change occurs, the needed actions are delegated asynchronously to other active objects, which also are implemented so that their execution time per an execution slot from the active scheduler is minimized. Running all this in the same thread isn’t maybe a theoretically perfect solution, but as the delays in the mobile call handling are relatively long, it was estimated to work well. If the server had been used to serve also the application UI, it would have potentially created additional delays into call handling, which also favored using separate engines for both activities.

Supportability requirements are quite well served already in the non-UI parts by using GRASPs in the design phase. By separating concerns and accessing engine functionality via facade interface it was possible to adapt the system by changing only some parts. The application UI had some supportability related requirements stated in the Vision artifact, namely internationalization and skin support. Internationalization is a process of enabling application localization, which in this case meant using strings from resource files. This way the locale can be changed by changing the used resource file without re-compiling the binaries. Skin support is a mechanism of enabling run-time look-and-feel configurability by providing new bitmaps, color schemes, and such in a platform-defined way. Both of these requirements were significant in that sense that they are usually required from all UIs and have to be considered early on.

The remaining ASRs go to the ‘+’ or others –category. One of the most concerning issues in the Inception phase was the publicity of the work, which can be categorized as a legal issue, as at least some of it was supposed to be documented in the public form of this thesis. The separation of concerns architectural principle was used to solve this issue by using publicity as the concern. In this process the engine subsystem emerged as the part that could be implemented without exposing any Series 90 platform proprietary technologies. This can also be seen as utilizing Protected Variations GRASP, as the S90 SDK was going to be published at some point anyway, but the schedules and contents might always change. In the end, the biggest impact this issue had on the software was that the profile handling wasn’t done via the engine interface.

Also implementation related constraints were identified during the project. Some are self-evident, but can still have a major impact on the architecture, e.g. that the application was targeted on Series 90 platform, so Symbian OS usage is also implied. The resource issues were also an implementation constraint, as missing knowledge clearly restricted some options out at some point. Later on it was too risky to change the design, thus also the time axis had architectural significance.

4.2 Logical View

The logical architecture view shows the conceptual organization of the software. Software can be organized in terms of the most important layers, packages, subsystems, frameworks, classes, and interfaces. It is useful to show also some external subsystems in the logical view to give the developed software a context. Figure 14 illustrates ReplyMate in the context of the external subsystems it uses and their interfaces. It is also encouraged to explain the diagram items shortly, because majority of the SAD readers are not experts in the handled area, at least not yet.

Series 90 UI framework provides the Symbian OS application UI framework and the framework for the notifier. Skin framework is used for implementing skin support. App Services and App Engines provide engine functionality for accessing the event log and the contacts database. Profile UI has an engine for accessing profile-related information. Messaging includes message server interface that provides access to the message store and SMS Client MTM that is needed for sending SMS message. Telephony subsystem is used for accessing the telephone functionality in the phone; in this case also mobile phone capabilities are needed, thus Multimode API is used. DBMS is used for accessing general database functionality.

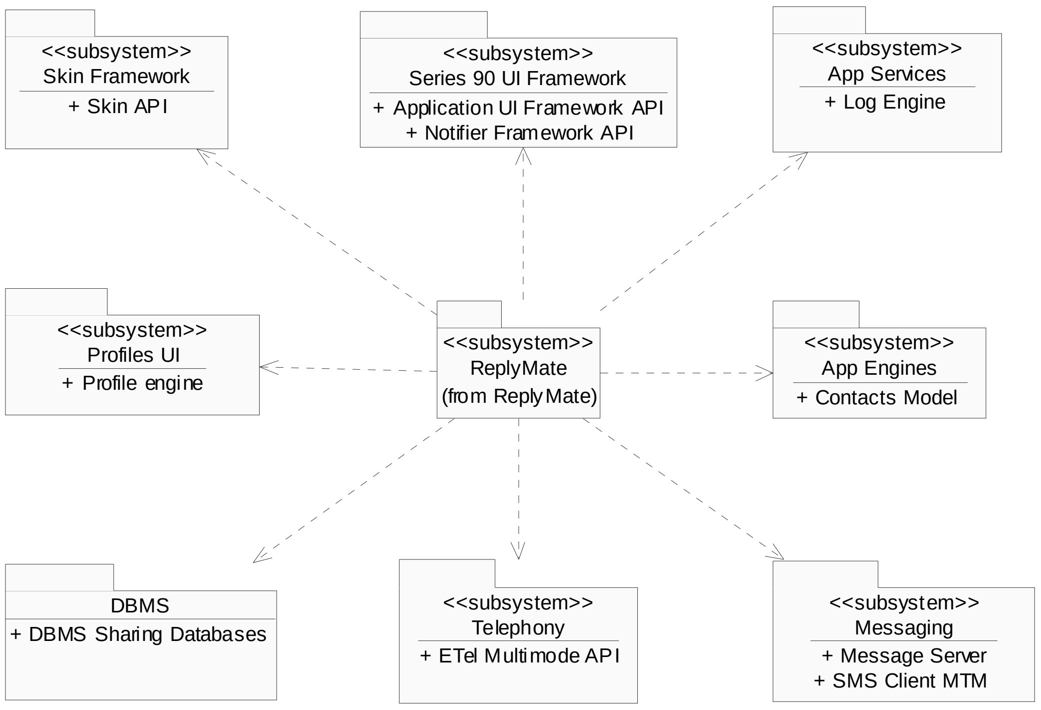


Figure 14: External subsystems used by ReplyMate.

The Layers architectural pattern [3] is a common way of dividing software into different layers based on their related functionality and level of abstraction. In this case some layers were identified, but no clear division efforts were made, as there was no real need. The area was still relatively small, thus the subsystem organization illustrated in Figure 15 was found to be enough. Still, the identified layers were used for organizing the diagrams where it was found to be descriptive.

Figure 15 illustrates the internal subsystems of ReplyMate and their dependencies. Both the server and the application UI use the engine for accessing the rules information. The application UI uses the server API for checking its existence and shutting it down. The server uses the notifier to give the user a possibility for canceling the message sending.

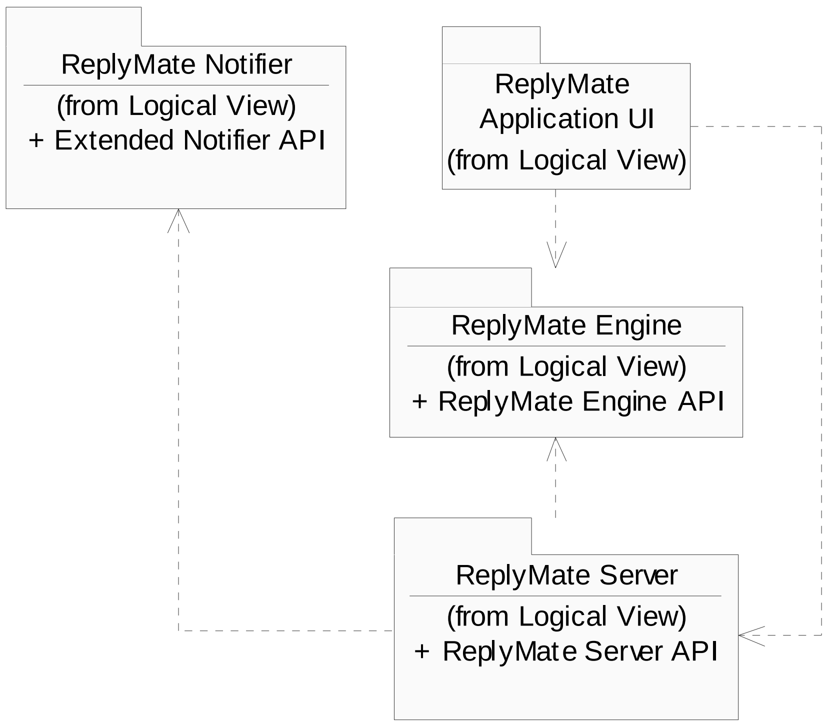


Figure 15: Internal subsystems of ReplyMate.

The class hierarchy of the ReplyMate server is illustrated in Figure 16. Also the sources for dependencies to other ReplyMate subsystems are shown. *CReplyMateLineObserver* acts as the main controller and owns most other objects. The Observer pattern [6] is widely utilized for controlling the state changes. The semantics are luckily simplified a bit by the Symbian OS Active Object idiom, which restricts the focus of control to one place at a time in the server thread.

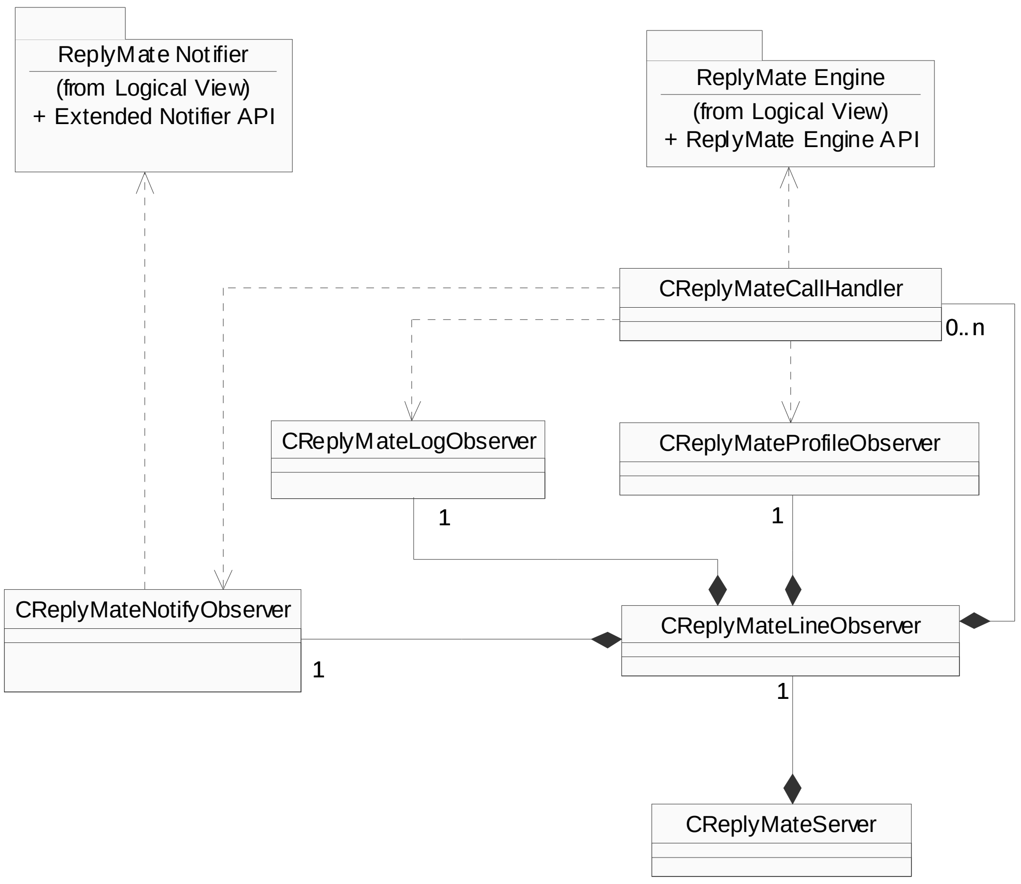


Figure 16: Class diagram of ReplyMate server.

The dynamic behavior of the server is illustrated in Figure 17, which shows a collaboration diagram of the Handle call use-case. It starts from the detection of state change in the phone line, which is signaled to *CReplyMateCallHandler* with the Ringing message. It gives the call handler the signal to activate the notifier that starts displaying information notifications. Then the current profile is fetched and the rules checked against it and the caller's number. A match is found and the corresponding message title is returned and delivered to be shown with information prints. When the line is disconnected from the ringing state, this is signaled to the call handler, which deactivates the notifier. Then a short time-out state is entered, during which the log observer signals about a missed call and the engine is signaled to reply the mate.

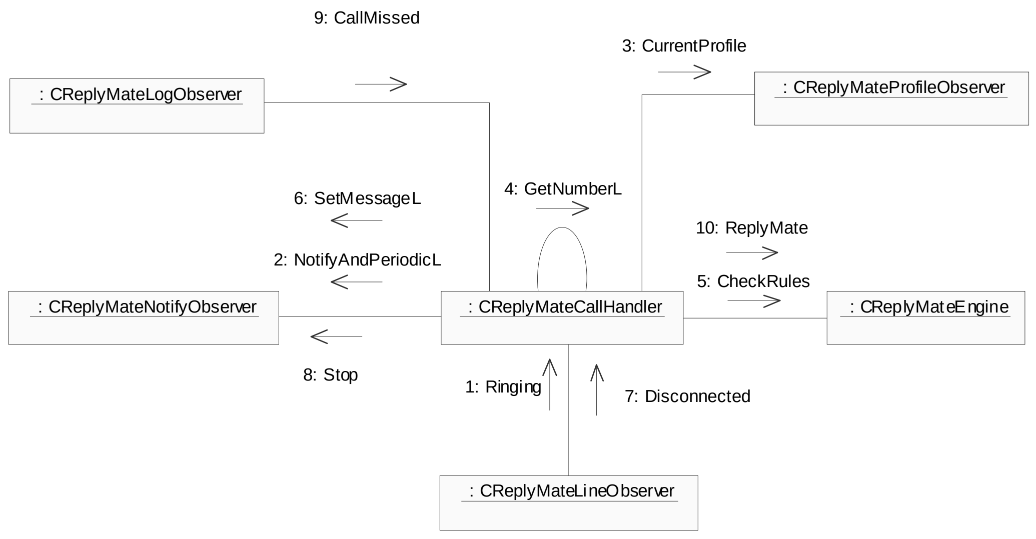


Figure 17: Collaboration diagram of Handle call.

Figure 18 shows a bit simplified class diagram of the ReplyMate application UI. The *CreplyMateRulesWizard* related dialogs are not shown, as they would only litter the diagrams without adding value. The Model-View-Controller architectural pattern [3] can be seen here: *CreplyMateEngine* is the model, *CreplyMateRulesView* is the view, and the *CReplyMateAppUi* is the controller. The application framework constructs the application by calling the defined method in the *CReplyMateApplication*, which constructs and returns a *CReplyMateDocument* to the framework. Then the framework calls the *CReplyMateDocument* object to create the *CReplyMateAppUi*. The controller then receives messages from the framework via *HandleCommandL* calls. These messages are usually generated by the user's menu accesses or button presses.

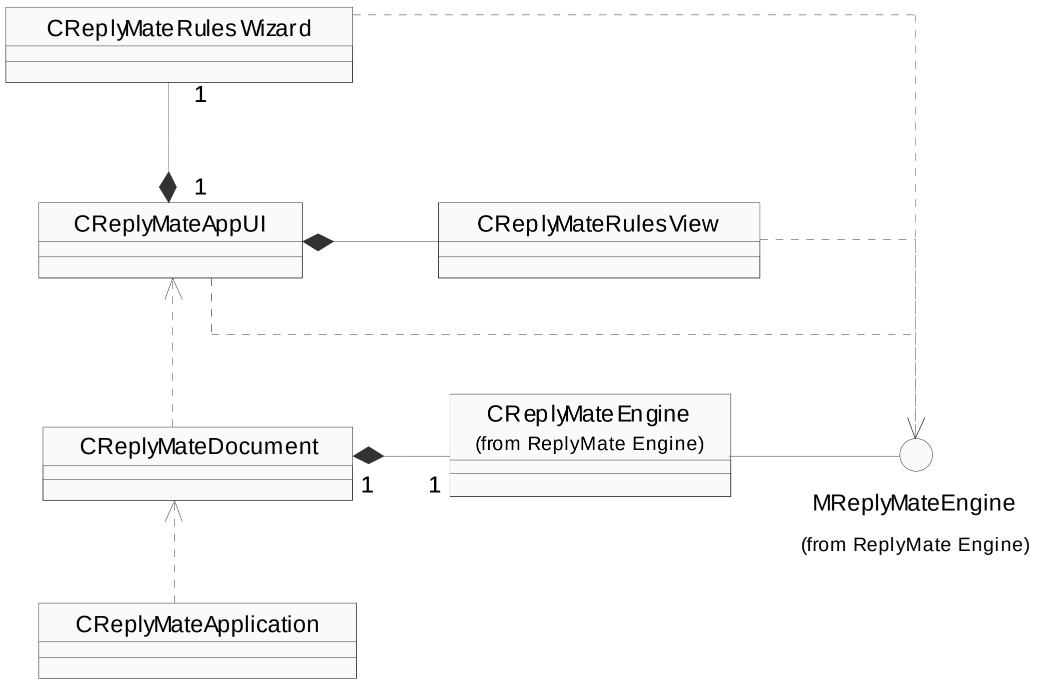


Figure 18: Class diagram of ReplyMate application UI.

Figure 19 shows a sequence diagram illustrating the Add rule use-case. It starts when the application framework calls the *CReplyMateAppUi* instance with a predefined value for adding rules. The controller then activates the rules wizard, which via different dialogs queries the contents for the rule, saves the input with the engine and finally commits the changes, when the engine saves the changes to the database.

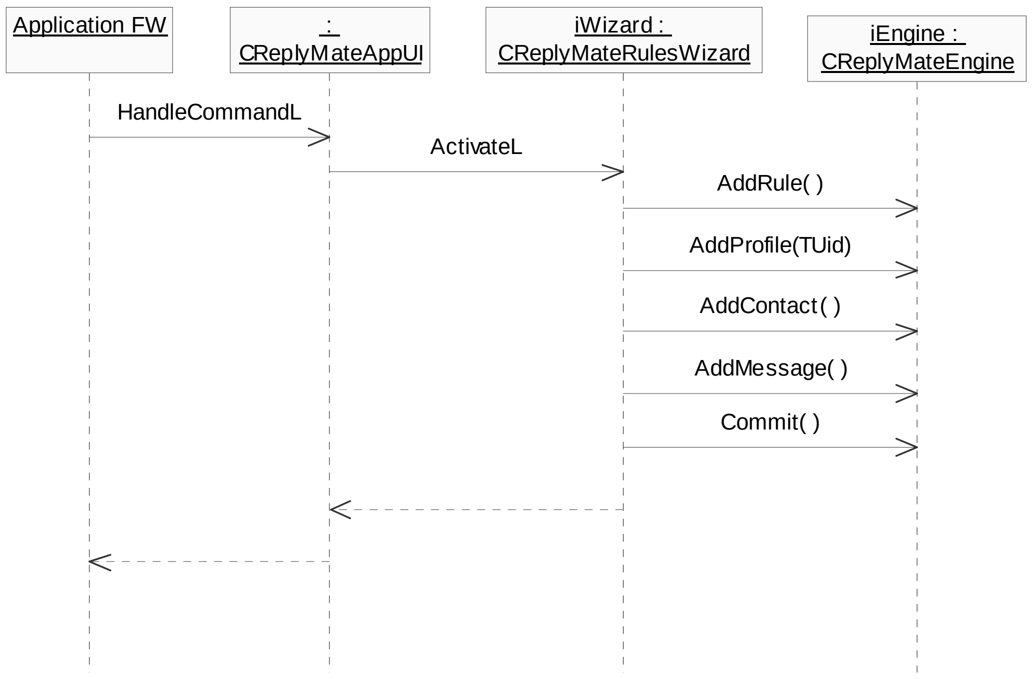


Figure 19: Sequence diagram of Add rule.

4.3 Process View

The process view shows the run-time allocation of the logical elements to the processes and threads, and also their responsibilities and collaborations. Figure 20 illustrates what different processes and threads make ReplyMate alive. The application UI is started into its own process by the application framework. The server process is started during the notifier startup which takes place when the Uikon server is started. The application UI uses the client interface of the ReplyMate server, which has a direct dependency to the server via Symbian OS IPC mechanism. The server and UI have also a dependency through the engine database, which is accessed with DBMS server. The server is connected to the notifier via the Extended notifier framework and its server, which is also controlled by IPCs in the *RNotifier* client interface.

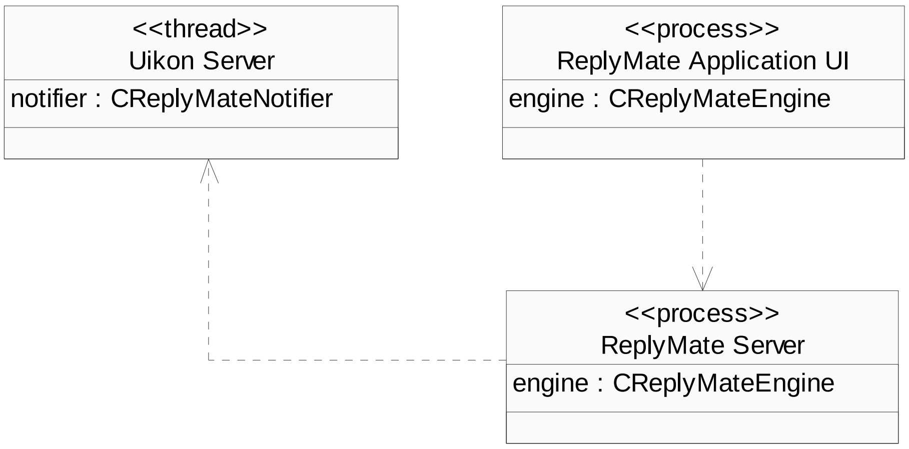


Figure 20: Process diagram of ReplyMate.

The engine database must be kept in synchronous state between the two processes. The DBMS server offers notifications which can be subscribed per database basis. When using this mechanism it's impossible to restrict the database update to the changed entries, the whole database must be reloaded. To minimize the file system accesses, a short wait period was implemented to the subscriber side. This way the database is only reloaded when no updates have occurred during the defined wait period; a value of one second was noted to be a good starting point. During that time the whole rule with its profile, contact, and message entries has typically been saved to the database. The mechanism is illustrated in Figure 21.

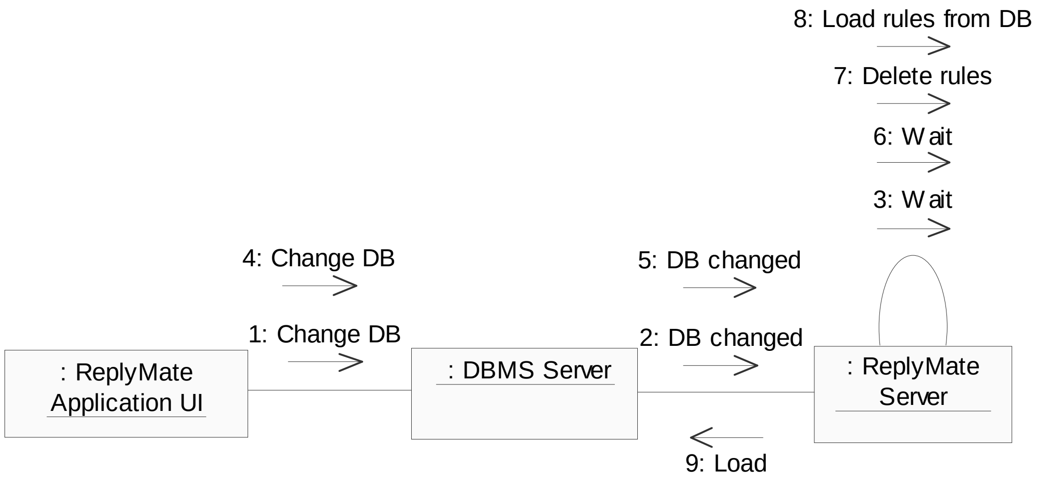


Figure 21: Collaboration diagram of engine synchronization.

4.4 Data View

The data view gives an overview of the persistent data schema; how the objects are mapped to persistent data and vice versa. Figure 22 shows the tables and their associations of the relational database created for ReplyMate. Every table has an auto-incremented primary key whose value is used as an object identifier in the engine according the Object Identifier pattern described in [20]. Profile and Contact have also a secondary key as a reference to the related rule. Additional identifier numbers point to entries in the corresponding data sources, e.g. contacts database. Profile and Contact have also a special “not” column for inverting their meaning. As the conceptual model showed only one-to-one relationship between rules and messages, the message information was included in Rule. A message has a title, which is by default the beginning of a message. In addition to the message information, Rule has a name and state of activeness.

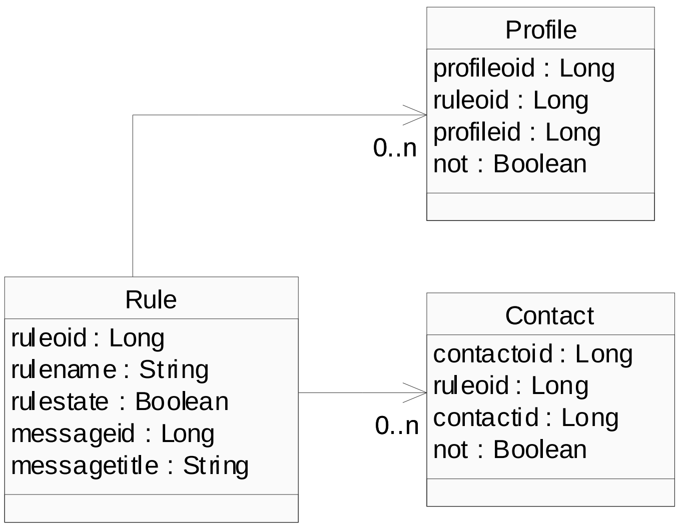


Figure 22: ReplyMate persistent data schema.

The engine uses a persistence facade class, *CReplyMatePersistenceFacade*, for storing and retrieving objects to and from the database. The Facade pattern [6] has been used to hide the database logic behind a common interface. The class acts also as a factory and creates the concrete database mappers; this is a simplification of the Abstract Factory pattern [6] by leaving some of the abstractness out. Every persistent class is derived from the *CReplyMatePersistenObject* class, which offers general persistence functionality. The Template Method [6] design pattern is used to enable more variability in the derived classes. Every persistent class implements the pure virtual *ObjectClass* method for identification as C++ doesn't have a mechanism for identifying the class of an object at run time. When an object is given to the *PutL* method of the facade, it identifies the object and forwards it to the corresponding mapper via *MReplyMateMapper* interface. The mapper inserts or updates the object to the database; zero object identifier identifies a new object. *CReplyMateDBMSOperations* class includes general DBMS related functionality, e.g. database initialization and opening. Put together, the implemented persistence functionality forms a persistence framework. Figure 23 illustrates the solution.

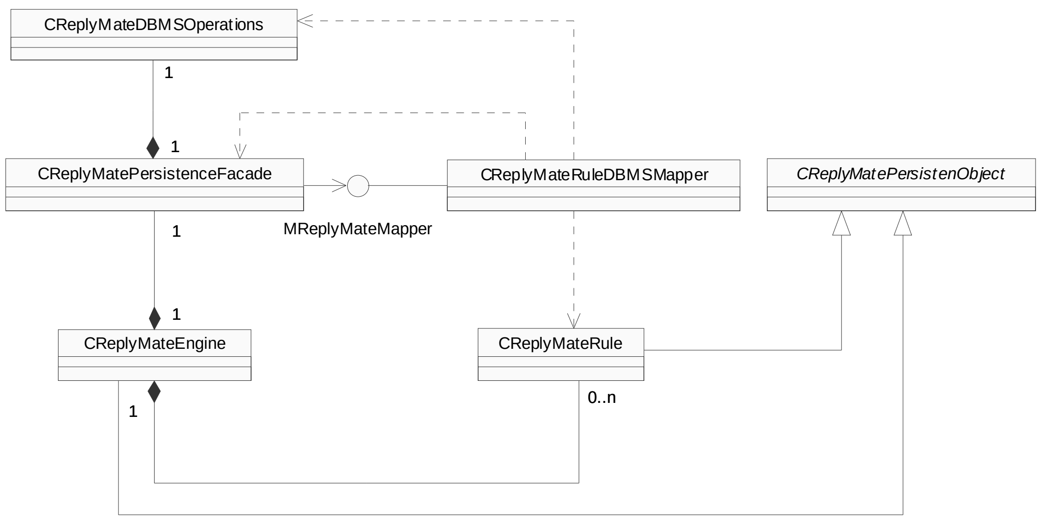


Figure 23: Persistence schema in ReplyMate engine for CReplyMateRule class.

5 MESSAGING APPLICATION ENGINE

Now that the progression of the work is handled and the application area has been explained, it is possible to explore one fruit of this work in more detail. The developed messaging application engine is a balance that was found between all the forces that have been explained previously. In practice these forces realized as a list of use cases that led into a list of features.

The use-case form of the requirements could be used with the whole application, but with middleware, such as the developed engine, they are less useful as there is no actor-system relationship. One could think, for example, the UI-engine relationship as one, but the used engine functionalities are rather atomic, there is no interaction between the parts, for example, when a profile is added to a rule. Thus, the requirements for the engine were derived from the use cases and presented as a list of features. Features are simply some things a system can do.

The design was carried out with the aid of the discovered industry best practices, such as patterns and general OOD principles. The static and dynamic aspects were modeled using the UML with the provided modeling tools. The implementation itself was conducted with standard Symbian OS tools and Nokia proprietary software packages, as no public and stable SDKs were yet available for the S90 platform. The engine could be almost thoroughly tested within the emulator with a set of unit and module test cases.

Even though different parts of the engine development are described here in a linear order, the development flowed through them within the corresponding UP disciplines. Maybe the implementation was a bit closer to the waterfall model than with the other parts as the use-case focus was lost in the feature list for a while; without broader experience it couldn’t be estimated if this normal with middleware.

5.1 Requirements

The functional requirements for the engine were derived from the use cases, so it was convenient to make the logical division for them based on the main use cases. As an exception, the profile changing was not handled in the engine; a separate utilities functional unit offered services for it. Figure 24 illustrates the identified requirements categories.

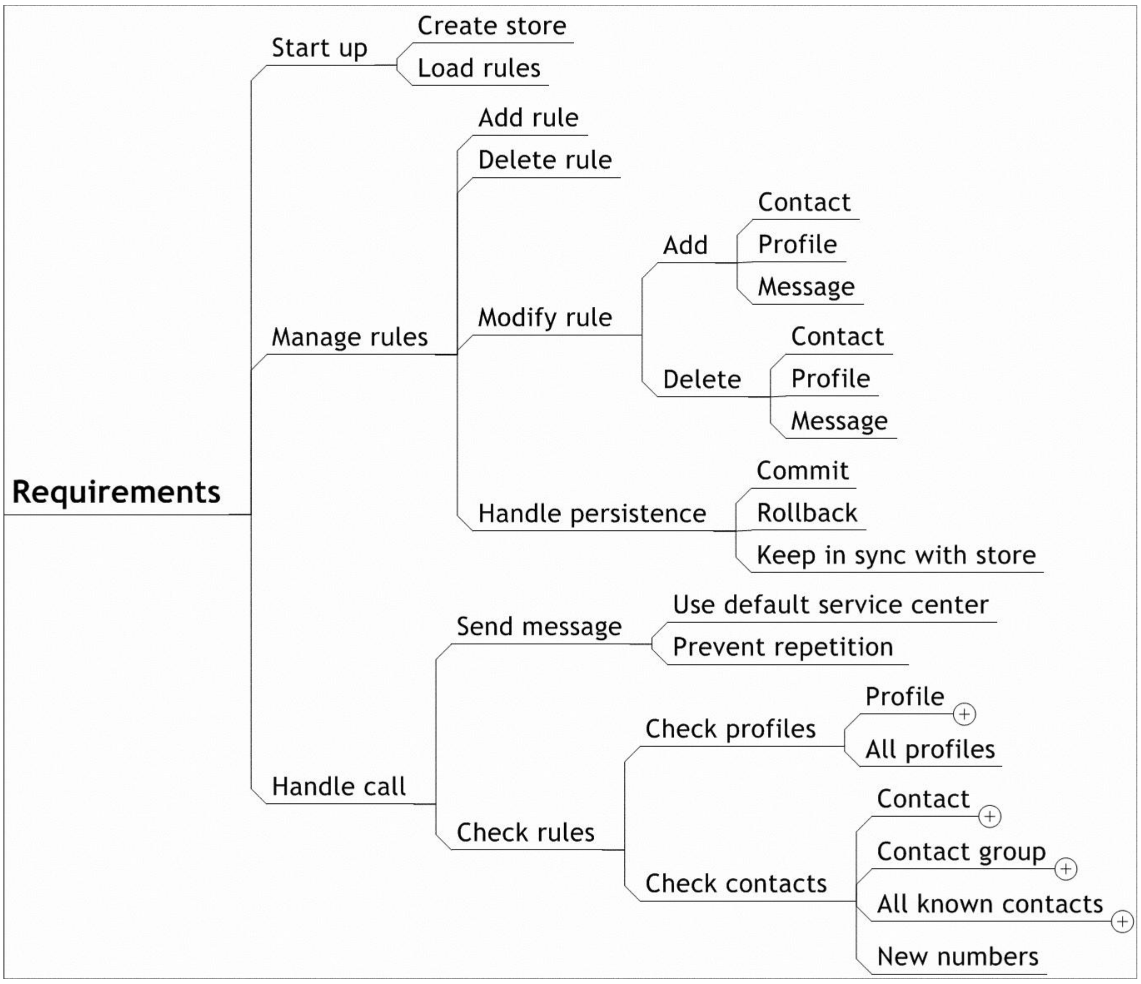


Figure 24: Requirements categorization for ReplyMate engine.

The categorization was managed with a mind-mapping tool, which was found to be natural for this kind of work. The UML use-case diagrams tended to get too crowded and it was more difficult to generate textual output from the used UML tools. Consequently, after requirements categorization a skeleton feature list for the engine could be generated from the items in the diagram. This way the features were also automatically divided into sub features continuing down to the leaf level.

In the feature list every item was given a unique identifier, an ordinal, and a short description. The ordinals sequenced the feature implementation in accordance with the use-case prioritization. The features were quite interdependent, so the ordering had to take into account the engine wide feature set with the use-case scenarios. For example, the message sending feature was needed in the first implemented use-case scenario, but it had no dependencies to other features, so it was given the first ordinal. On the other hand, the rule checking needed some rules to check, thus it and adding rules was given the same ordinal. The rule didn't need to be complete as long as it implemented the designed interface and allowed the scenario to be finished. Table 5 shows the feature list illustrating the described case.

Table 5: Example features for the engine.

Feature ID	Ordinal	Description
REPL_ENG_FE_1	1	The engine shall do message sending based on a given phone number and a message reference.
REPL_ENG_FE_2	2	The engine shall do rule checking based on the stored rules.
REPL_ENG_FE_3	2	The engine shall do rule adding.

All but the most low-level features have also sub features associated with them. The identifiers for sub features were decided to be in a dotted form representing their parent feature identifier and then their own. In this case the feature list was never managed in the list form, so there was no need for deciding whether to represent these features with their parents or in a separate list, both ways were possible depending on the need.

In addition to the features shown in Figure 24 also all non-functional ASRs were a part of the engine requirements. The reliability requirement was seen as the most important one and was marked for continuous attention during the design.

5.2 Design

The design work followed the order given by the use-case driven feature ordinals and UP iterations. The design could have been frozen at the earliest when the feature list was complete, which it was in the beginning of the third Elaboration iteration. Instead, in the UP fashion, the engine design lived until the end of the Elaboration phase. As there was also a strong ongoing learning process involved, the stream of new design ideas seemed to be endless. All possible patterns and design principles lured to try them out, but finally this was ended by the restricted useable time. The final design, without some already handled parts of persistence, is illustrated in Figure 25.

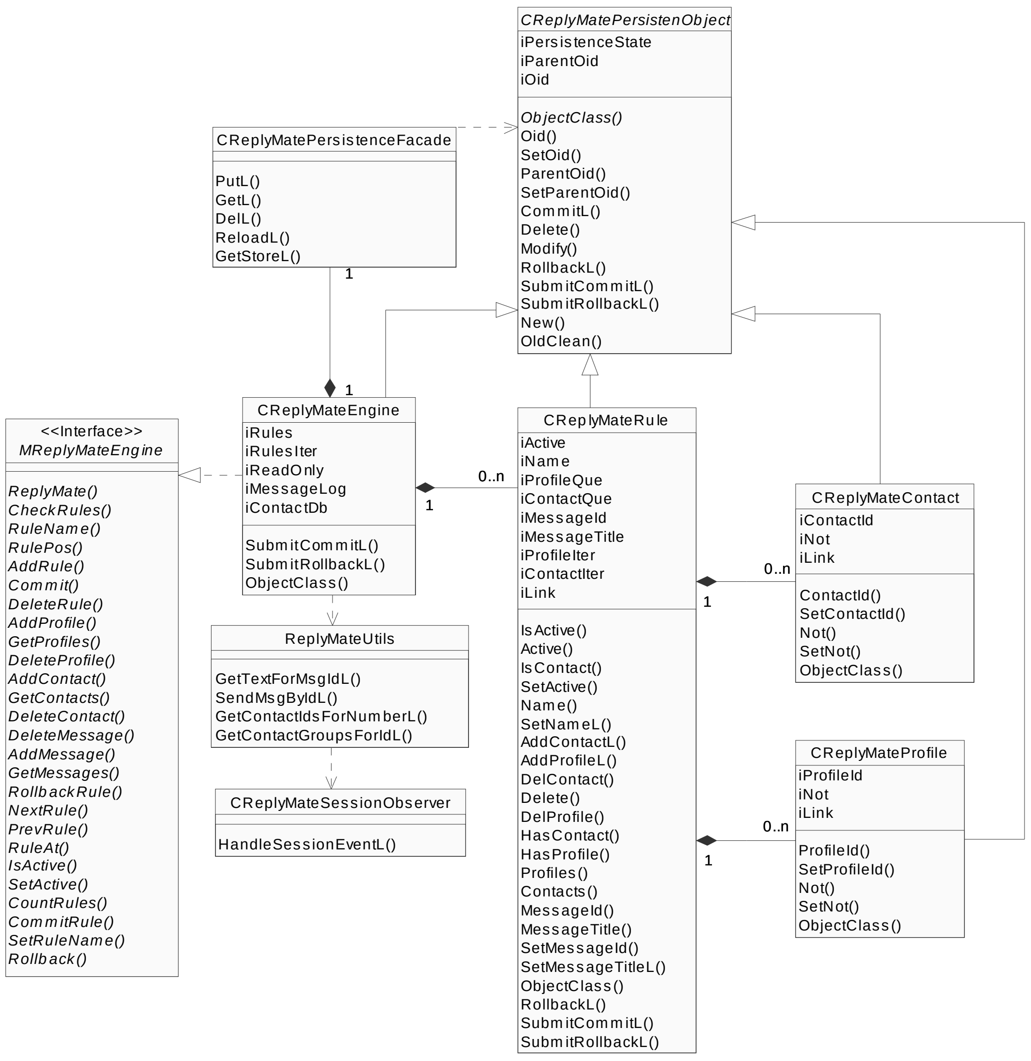


Figure 25: ReplyMate engine DCD.

Starting from the left side, the engine interface *MReplyMateEngine* lists all externally accessible functions. The function signatures are left out here and in the most other diagrams for brevity; visual modeling with A4 sized pages is clearly an oxymoron. At first a multi-class interface was designed but its usage needed comprehensive knowledge about the engine internals. Therefore, the Facade [6] design pattern was applied and a common interface class created for the whole functionality. Some of the functions convey their idea already with their names; most others can be seen in action in the diagrams throughout this thesis. The concrete engine implementation is in the *CReplyMateEngine* class, which also holds the necessary attributes such as the doubly linked list of rules.

The static *ReplyMateUtils* class is the container for utility functions needed for serving the requests from the server. For faster response times, a handle for contacts database is kept open in the read-only engine that the server uses. Connection to the messaging server is needed less frequently, so it is opened only when needed. The engine also keeps a log of the sent messages, by default the information is kept at least for one hour. This makes it possible to prevent repeated message sending to a frequent caller.

The basic idea in the engine design is that it acts as a rule Iterator [6] for the UI and contributes the rule logic for the server. The requests, which target rules, are delegated to the current rule; the Information Expert GRASP is applied in the same manner throughout the engine. Thus, the rule, profile, and contact classed hold and utilize the information that is related to them. All of these classes are also derived from the same base class *CReplyMatePersistentObject*, which offers common persistence related functionality, such as state handling. Some of the functionality, for example submitting commit requests down the object hierarchy, isn’t common for all derived classes. Therefore, the Template Method [6] design pattern is utilized and by it the required additional functionality can be implemented in the derived classes.

The Singleton [6] design pattern is in a very central position within design patterns; many of them use it as a basis for their solution. For example, facades are usually also singletons. This creates some additional issues with the Symbian OS, because static data cannot be used in DLLs. At least one more general workaround exists [15], but in this case it was not estimated to be worthwhile and the needed references to facade class instances were spread during the construction procedures.

The static parts of the created persistence framework were already somewhat covered in the architecture portion, thus they were left out from the already too crowded diagram. The dynamic aspects of the implemented persistence solution are illustrated in Figure 26. Important things to note are the usage of *SubmitCommitL* to vary persistence behavior and how the objects are identified and then sent to the corresponding mapper.

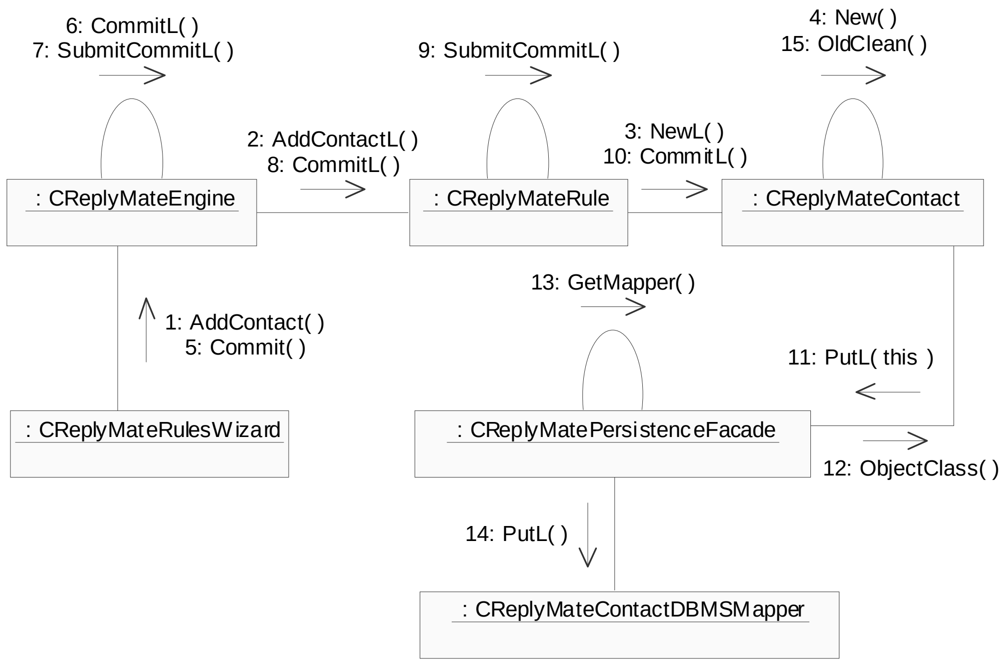


Figure 26: Collaboration diagram of contact adding.

When the state of a persistent object changes, also its persistence state must be changed accordingly. This state is then used when a commit or rollback is requested; the related state changes and transitions are shown in Figure 27.

The design requires disk accesses for the rollbacks, which is generally a bad thing in devices that use flash memory as mass storage. Therefore, also caching rollback functionality was planned, but it had to be dropped for schedule reasons.

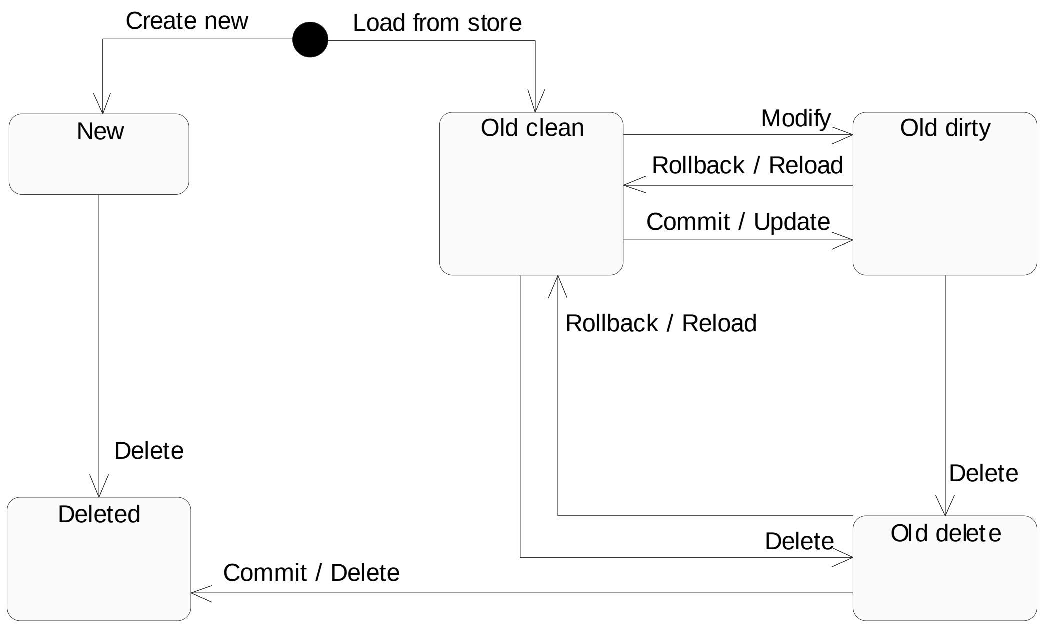


Figure 27: States and transitions in persistent objects.

Figure 28 illustrates the other side of the coin from the Figure 17, where the message number 5 starts this interaction. The *CheckRules* function of the engine facade is actually the only one which doesn't just delegate the responsibility to somewhere. First it gets the contact identifiers and the contact group identifiers from the contacts database with the of the *ReplyMate* class. If no contacts are found the number is a new one and is tested with wildcard contact for explicit denial, i.e. not all contacts –contact has this special meaning. After acquiring contacts the engine goes through all the rules and requests their activity and contact statuses against the current profile and the found contact identifiers. The rules return predefined bit patterns according to their properties. The patterns are combined with bitwise OR operation and the rule that get the biggest number is the winner, thus its message information is returned with the resulted bit pattern. The bit pattern is returned even if no matching rule is found; this is valuable information for testing.

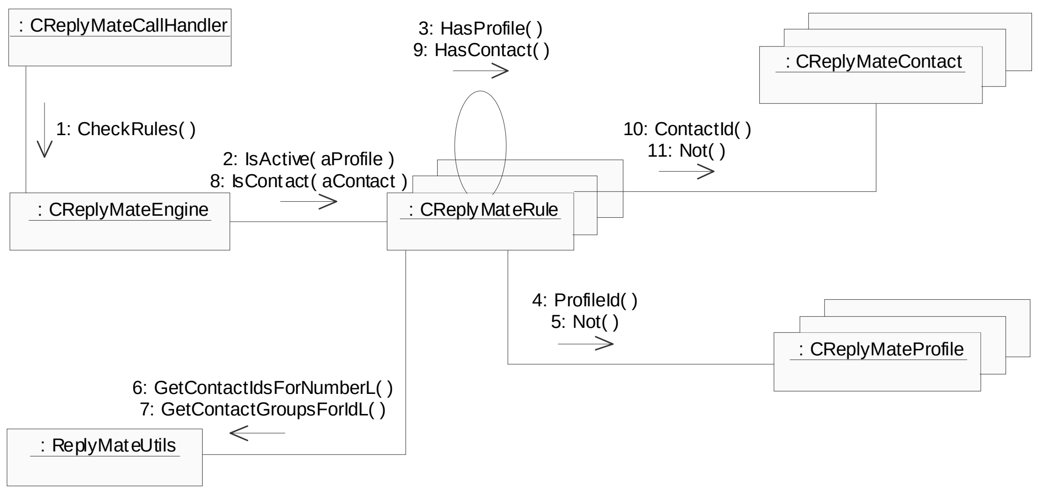


Figure 28: Collaboration diagram of checking rules.

The bit positions and their meanings are given in Figure 29. Wildcards mean that there is a contact or profile entry that matches all requests. Explicit matches always get priority over wildcards or contact groups, but explicit denials are the strongest rules. Thus, the order of the bits automatically handles the priorities; as an additional example, if there is an explicit denial for a contact group, an explicit contact match still overrides it. Active bit is always set if a profile or activity state match is made.

Meaning	Active	Contact explicit	Contact group	Contact wildcard	Contact not	Profile explicit	Profile wildcard	Profile not
Bit	128	64	32	16	8	4	2	1

Figure 29: Meanings of the bit positions in the returned bit patterns.

Figure 30 illustrates the rule checking a bit closer (pun intended) to the actual implementation. It can be seen how the rules are never tested more than needed, if a rule is not active by its own state or the current profile, it can be skipped.

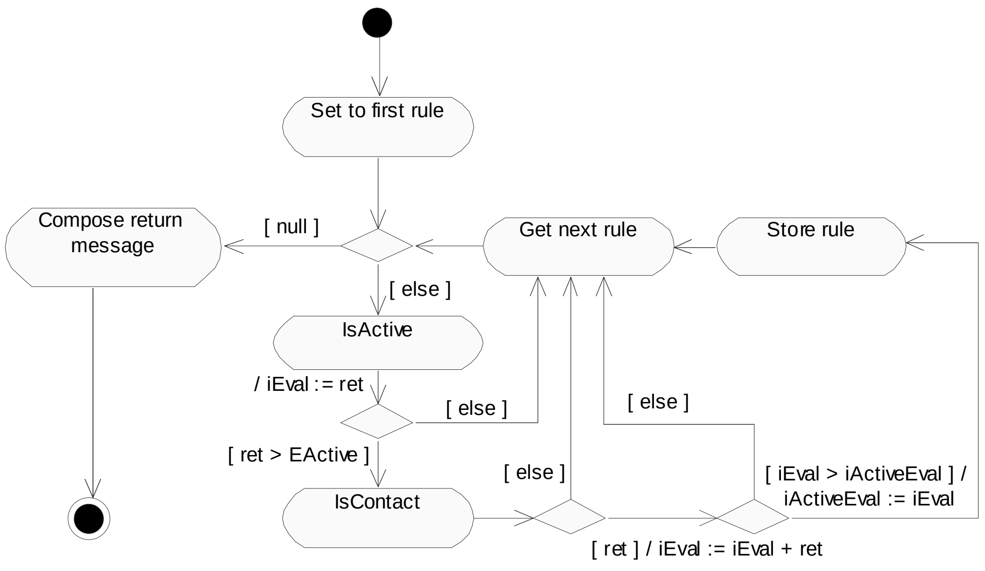


Figure 30: Activity diagram of rule checking.

After a proper rule has been found, the message must be sent. Figure 31 illustrates this rather complex operation. An outline of the events starts with connecting to the message server with *OpenSyncL*, which is followed by the fetch and set up of the SMS client Message Type Module, or MTM, from the MTM registry. Then the given message store index is used to initialize the correct server entry, which is copied just in case it was being edited, then the copy is set up with the correct settings. This includes removing the possible previous receivers and adding the new one. When the set up is finalized the message is saved and moved to outbox, from where it is scheduled for sending.

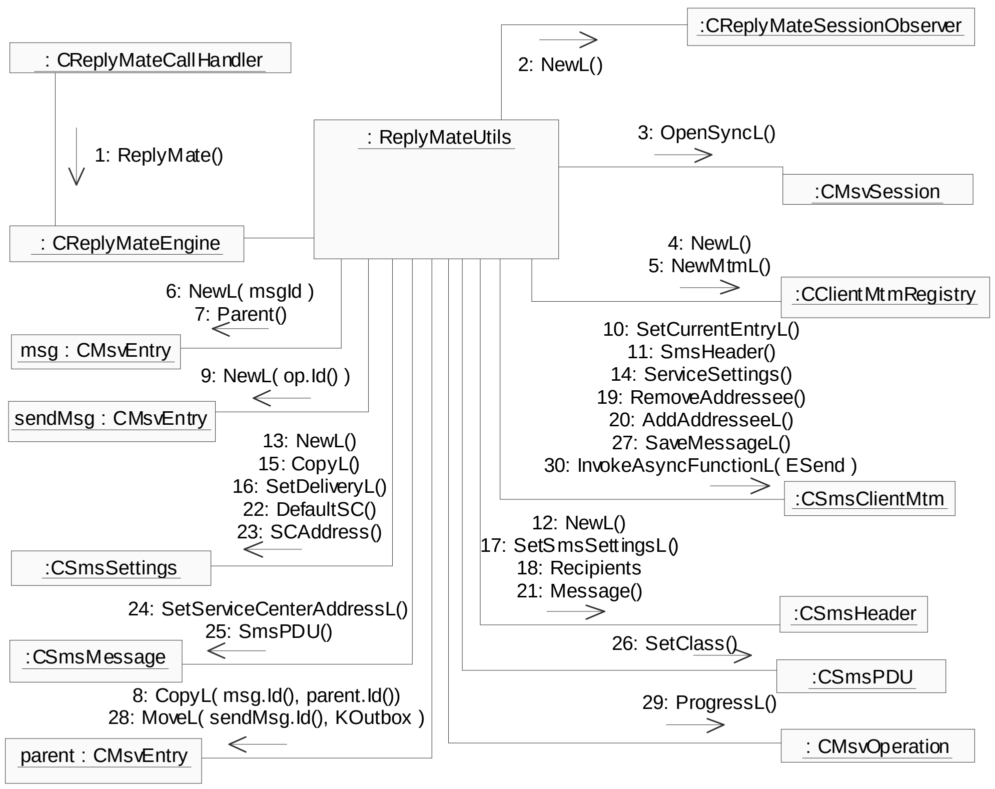


Figure 31: Collaboration diagram of message sending.

5.3 Implementation

The implementation was carried out with the standard Symbian OS tool chain, which experienced some changes during the development. In the beginning Microsoft Visual Studio was used and later on changed to MetroWerks CodeWarrior. This required small changes to the build information files and sources to take the new compiler into account and because it was stricter in many issues. In particular, const correctness checking was getting more attention. In addition to the changes in the internal workings, the change involved also major changes to the GUI. These had little effect on the project, because strong bonds to the old tools were not yet established.

The code base was managed with modern version control tools, which were in many aspects too heavy and complicated for a project of this size. There was previous experience of a more practical version control program, namely Concurrent Versions System [4], or CVS, and finally the change resistance lead into creating a migration path. Thus, the daily changes were kept in a CVS repository and its contents were exported to the version control system of the company on a per week basis.

The implementation work itself proceeded in the UP boundaries, but with additional focus during a couple of iterations. The work was mostly conducted during the Elaboration phase, as the engine was seen as an important architectural part and so its interfaces should be as ready as possible when the Construction phase was started. Some minor changes occurred even after that; it's hard to commit to keeping interfaces frozen if no one else is dependant on them.

Some useful software project management issues, such as collecting software metrics during the project, got unfortunately attention only when their results would have been useful. These would have created nice information for reporting, especially when combined with the UP disciplines. Now these project issues were too easily neglected in favor of learning and design. In general, the utilized UP references handled practical project management issues very briefly, so maybe an additional reference handling these issues together would have been useful. Lesson learned.

Consequently, the only metrics that are available reflect the final situation. With a quick search [7] the area of OO software metrics hasn't yet stabilized; a few suggested metrics, which were easily available, are listed in Table 6.

Table 6: Identified software metrics for ReplyMate engine.

Metric	Result
Amount of source code	~ 30 kB
Average functions per class	10.6
Amount of classes	10
Amount of exported functions	27

The list of metrics is a bit lacking because the used design tools didn't support automatic calculation of some more descriptive metrics such as cohesion and coupling. Required research for algorithms to manually produce more metrics was out of scope for the context.

Quality control in the planned form of peer reviews was waiting for the Construction phase to end. The project never reached that milestone and there were no concrete business reasons driving the quality issues, thus the reviews were never held. The effects this had on quality are hard to estimate, but based on previous experiences the reviews discover only minor issues as usually no one has time to deeply focus to the handled area. After all, the most important artifact of any software development - functional binaries – was produced. The engine DLL implemented the feature set that was planned for it and enabled further development of the application UI and server parts.

5.4 Testing

The engine testing followed the already described method of planning and writing test cases before the tested part of the engine was written. This method seems to have only good sides to it; it obviously raises quality awareness as the testing cannot be forgotten after the implementation seemingly is in working condition. In addition, the argument of subjective feeling of achievement actually realized, i.e. it was rewarding to get through the test cases.

The test cases were implemented with a console application with the aid of Symbian OS testing functionality that is based on using the *RTest* class. It automatically creates a console screen for printing the results of the test cases. The test cases can be nested and named; the names are printed when a new level of nesting is started. In addition to console printing, all output is also directed to the debugging output, so it can be monitored from the IDE or the debug port on a device. It is also advisable to wrap the main testing function against leaves so that the leave error codes can be caught and printed in a controlled way. Further, still another level of wrapping is useful to be done with Symbian OS heap balance checking macros, which detect possible memory leaks.

Some of the inner classes of the engine had to be compiled into the testing software to provide unit testing capabilities for them. The persistence functionality was the most important unit tested part; other areas were mostly covered with module test cases that used only the external DLL interfaces. Table 7 lists one of these test cases.

Table 7: Module test case for ReplyMate engine.

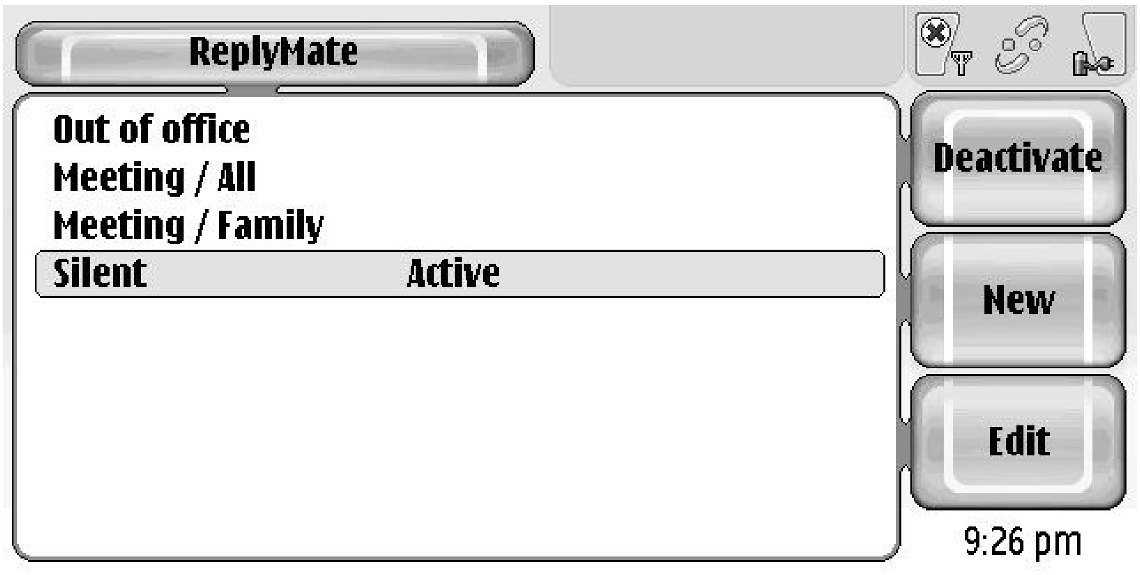
```
void CreateEmptyRule( CReplyMateEngine* aEng )
      {
      _LIT( KCreateEmptyRule, "Empty rule creation test.." );
      test.Start( KCreateEmptyRule );

      test( aEng->AddRule() == KErrNone );

      test( aEng->Commit() == KErrNone );

      test.End();
      }
```

After a test case fails the reasons for it must be discovered. The emulator and all software running on it are actually Windows executables and debugging them is well supported by the IDE. Therefore, the engine could be tested and debugged within the emulator; even the message sending capabilities, because in practice the message is just copied into the outbox folder of the device and then scheduled for sending. Without real hardware the message just doesn’t leave anywhere, because the lower layers of the messaging subsystems are missing. In contrast, the server was harder nut to crack, as it listened for telephone events. Thus, also some in-circuit debugging tools were utilized with it.

One important metric of testing is the code coverage, i.e. the percentage of the code that is executed during the testing. The target for this should always be 100%, but it is difficult to achieve, as some rare error conditions cannot always be created artificially. There are many code analysis tools that also check the code coverage, but in this case the coverage was estimated manually as the area was still relatively small. The error situations are handled with the Symbian OS leave mechanism, the possible leaves are trapped in the engine facade and the corresponding leave codes are returned to the host program. This way the error handling responsibility was delegated away from the engine and the leaves could be handled with common code. Thus, it could be covered by the test cases easily by creating an artificial out-of-memory situation, which is possible with the emulator. Therefore, the achieved code coverage was estimated to be very near 100%.

6 MESSAGING APPLICATION USER INTERFACE CONCEPT

The application user interface work was halted during the first construction phase, so the UI was not finalized. Thus, the results presented here aren't too polished, but they give some idea of the look and feel of the application. Figure 32 shows the rule editor with a few rules and the touch screen command buttons for managing them. More actions, such as delete and undo, can be launched from the menu, which is opened from the top left bar with the ReplyMate title.



Out of office
Meeting / All
Meeting / Family
Silent Active

Figure 32: ReplyMate rule editor main view.

Figure 33 illustrates the UI parts that the server launches when it detects a call. In a device there would be the telephone application in the background, but it isn't available in the emulator. The information print shows information about the ongoing action, in this case the Meeting rule has caught the call and its title is showed. In this case the user has apparently used the beginning of the message as its title, which option is offered automatically by the rule editor.

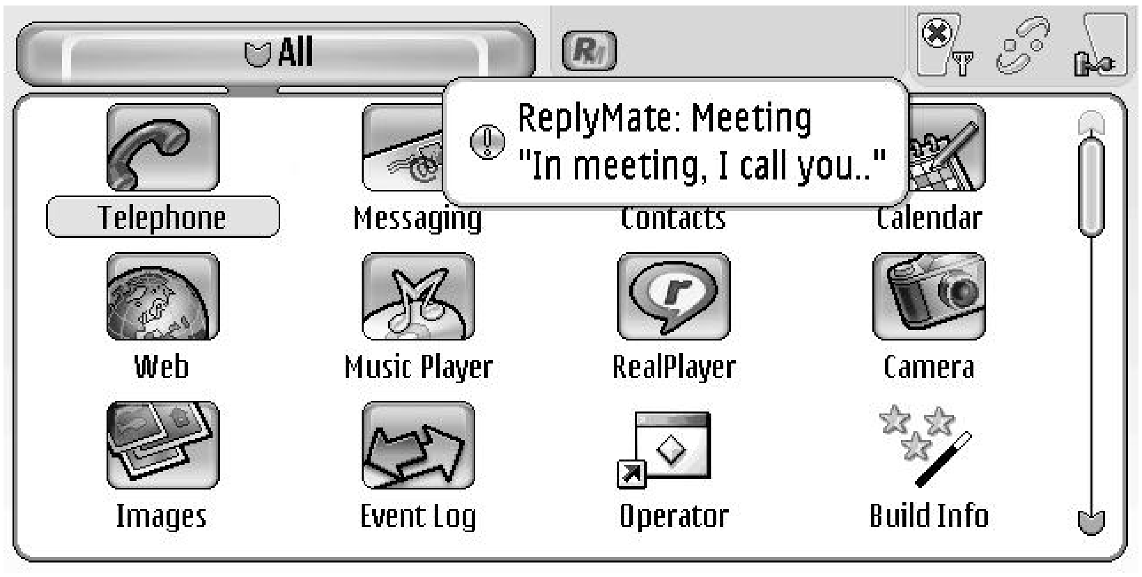

ReplyMate: Meeting
"In meeting, I call you.."

Figure 33: Call handling in action.

7 CONCLUSIONS

The developed messaging application engine is a sum of different forces. Some other balances between them probably could have been found even with the same industry best practices which were applied now. If these practices are the science of software development then the art must be in finding the balance.

The Unified Process guided the development. This also implies usage of the Unified Modeling Language and patterns for visualizing abstract issues and harnessing legacy design knowledge. All these were found to be worthy even in a project of this size; thus they further augmented their position in the author’s toolbox.

The application engine features were driven, and proven, by implementing a messaging application, which responds to missed calls with short message service messages. Multiple system services were used, and acquainted with, in the process. Thus, also the learning targets were achieved and even exceeded in many areas.

The resulting messaging application engine fulfilled the specified non-functional goals of mitigated resource usage and possibility of reuse. At least on paper, as the engine was designed to harness the available platform services and to be as easy to use as possible. The reusability of a component is proven only by reuse, which may realize at least as a learning tool also in the future.

The surfaced improvement ideas involve added abstraction; at least other message types could be supported. Further, the software could evolve into a more general event handling software, which would respond to events with actions. This could lead into creation of an agent application for mobile devices and it does sound as an interesting idea.

REFERENCES

[1] Beck, Kent. Extreme Programming Explained: Embrace Change. Addison-Wesley, 2000. ISBN 201-61641-6.

[2] Booch, Grady. The Unified Modeling Language User Guide. Addison-Wesley, 1998. ISBN 0-201-57168-4.

[3] Buchman, Frank. Pattern Oriented Software Architecture: A System of Patterns. John Wiley & Sons, 1996. ISBN-0-471-95869-7.

[4] Concurrent Versions System. World Wide Web pages. Available: http://www.cvshome.org. [Referenced: 21.04.2004]

[5] Digia, Inc. Programming for the Series 60 Platform and Symbian OS. John Wiley & Sons, 2003. ISBN 0-470-84948-7.

[6] Gamma, Erich. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[7] Google. Search engine. Parameters: "object oriented" "software metrics". Available: http://www.google.com. [Referenced: 20.04.2004]

[8] Grady, Robert. Practical Software Metrics for Project Management and Process Improvement. Prentice-Hall 1992. ISBN 0-137-20384-5.

[9] Harrison, Richard. Symbian OS C++ for Mobile Phones. John Wiley & Sons, 2003. ISBN 0-470-85611-4.

[10] IBM Corporation. World Wide Web pages. Available: http://www-306.ibm.com/software/rational/ [Referenced: 21.04.2004]

[11] Jacobson, Ivar. The Unified Software Development Process. Addison-Wesley, 1999. ISBN 0-201-57169-2.

[12] Larman, Craig. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd ed. Prentice Hall, 2002. ISBN 0-13-092569-1.

[13] Nokia Corporation. Forum Nokia. Available: http://www.forum.nokia.com/main.html [Referenced: 21.3.2004]

[14] Sanders, Peter. Creating Symbian OS phones. Revision 1.1, April 2002. Available: http://www.symbian.com/technology/create-symb-OS-phones.html. [Referenced 24.11.2003]

[15] Sukanen, Jari. Master’s Thesis: Extension Framework for Symbian Applications. Helsinki University of Technology: Department of Computer Science and Engineering, 2004.

[16] Symbian Ltd. Symbian Developer Library v7.0s. Available: http://www.symbian.com/developer/techlib/v70sdocs/doc_source/index.html [Referenced: 29.2.2004].

[17] Symbian Ltd. Symbian Developer Network. Available: http://www.symbian.com/developer/index.html [Referenced: 08.03.2004]

[18] Tasker, Martin. Professional Symbian Programming: Mobile Solutions on the EPOC Platform. Wrox Press, 2000. ISBN 1-861003-03-X.

[19] Texas Instruments. Product Bulletin for OMAP1510. TI Inc. 2001. Available: http://focus.ti.com/pdfs/vf/wireless/omap1510_bulltn.pdf [Referenced: 24.11.2003]

[20] Vlissides, John**.** Pattern Languages of Program Design 2. Addison-Wesley, 1996. ISBN 0-201-89527-7.