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

Design Patterns in EPOC Software Development

The topic of the Master's Thesis has been accepted September 13, 2000 by the Department Council Meeting of the Department of Information Technology.

Supervisor:
Prof. Heikki Kälviäinen

Instructor:
M.Sc. Jouni Vaahtera

Lappeenranta April 1, 2001

Kimmo Hoikka
Pellonmäenraitti 3 as 8
53850 Lappeenranta
+358 40 738 0747
Kimmo.Hoikka@Digia.com

ABSTRACT

Lappeenranta University of Technology
Department of Information Technology
Kimmo Hoikka
Design Patterns in EPOC Software Development
Master's Thesis, 2001
81 pages, 31 figures
Supervisor: Professor Heikki Kälviäinen
Keywords: Design Patterns, Architectural Patterns, Design Principles, OOD, EPOC, UML

The thesis studies design patterns in the EPOC operating system. The thesis studies general design patterns and also patterns in the EPOC operating system. The focus is on the requirements and the benefits of using design patterns in EPOC along with its own patterns. During the thesis an EPOC software was designed using design patterns and following the design principles.

Design patterns have become more common in the recent years. The basis for the design patterns is the design principles and environment specific principles. Design patterns are a part of software pattern family, which contains process, analysis, architectural, etc patterns. Design patterns speed up and simplify the design, and improve reusability in higher abstraction level.

EPOC is one of the most common operating system for the future mobile environments. EPOC is completely object-oriented and contains several patterns that must be understood by software developers. Since the platforms where EPOC is mostly used have limitations on resources, the developers must be careful when applying general design patterns into EPOC. Some general patterns must be modified to fit into EPOC and some do not work at all.

TIIVISTELMÄ

Lappeenrannan teknillinen korkeakoulu
Tietotekniikan osasto
Kimmo Hoikka
Design Patterns in EPOC Software Development
Diplomityö, 2001
81 sivua, 31 kuvaa
Tarkastaja: Professori Heikki Kälviäinen
Hakusanat: Suunnittelumallit, arkkitehtuurimallit, suunnitteluperiaatteet, oliopohjainen sunnittelu, EPOC, UML
Keywords: Design Patterns, Architectural Patterns, Design Principles, OOD, EPOC, UML

Työssä tutkittiin oliosuunnittelumalleja EPOC-käyttöjärjestelmässä. Työssä tutkittiin sekä yleisiä suunnittelumalleja että EPOC-ympäristössä esiintyviä oliorakenteita, niiden aiheuttamia vaatimuksia sovelluksille sekä niiden käyttämisestä saatavia hyötyjä. Työssä toteutettiin EPOC-ohjelmiston suunnittelu hyödyntäen suunnittelumalleja ja periaatteita.

Oliosuunnittelumallit ovat yleistyneet huomattavasti viime vuosina. Suunnittelumallien lähtökohtana ovat sekä yleiset että ympäristökohtaiset suunnitteluperiaatteet ja säännöt. Suunnittelumallit ovat osa isompaa rakennekokonaisuutta, joka käsittää sekä prosessi-, analyysi-, arkkitehtuuri- ym. malleja. Oliosuunnittelumallit nopeuttavat ja helpottavat suunnittelua sekä parantavat uudelleenkäytettävyyttä korkeammalla abstraktiotasolla.

EPOC on tulevaisuuden mobiililaitteiden yleisimpiä käyttöjärjestelmiä. EPOC on kokonaisuudessaan oliopohjainen ja sisältää lukuisia oliorakenteita, joiden ymmärtäminen on sovelluskehityksen kannalta elintärkeää. Koska ympäristöt, joissa EPOC-käyttöjärjestelmää käytetään, ovat yleensä resurssien puolesta rajoittuneita, on yleisten suunnittelumallien käytössä oltava tarkkana. EPOC vaatii yleisiin suunnittelumalleihin muutoksia ja estää joidenkin käytön kokonaan.

PREFACE

This thesis was written for Digital Information Architects, Digia Inc. in Digia Lappeenranta branch office. I started to get acquainted with design patterns early in spring 2000. The implementation part of this thesis was done during autumn 2000 and the writing started at September 2000. I also had the opportunity to give a presentation on the subject at the Symbian Developer Expo November 2000.

I would like to thank the people who have helped in processing this thesis during its lifetime. Especially the instructor of the work, Jouni Vaahtera, whose idea this subject was in the first place. I must also thank the software architects at Symbian, whom I had the opportunity to talk with during the Symbian Developer Expo in November 2000. They gave an important insight into EPOC and its structures. Big thank must also go to the supervisor of this thesis, Heikki Kälviäinen about good guidance and patience on continuous delays on deliveries ☺.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

ADP	Acyclic Dependency Principle
API	Application Programming Interface
APPARC	Application Architecture
DFRD	Device Family Reference Design
DIP	Dependency Inversion Principle
DLL	Dynamic Link Library
EXE	Executable program
GCC	Gnu C Compiler
GDI	Graphics Device Interface
GOF	Gang-Of-Four
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
ISP	Interface Segregation Principle
LAF	Look and Feel
MVC	Model-View-Controller
OCP	Open Closed Principle
OOD	Object Oriented Design
OOM	Out Of Memory
OOR	Out Of Resources
OPL	Organizer Programming Language
PC	Personal Computer
PDA	Portable Digital Assistant
RAD	Rapid Application Development
RAM	Random Access Memory
ROM	Read-Only Memory
RTTI	Real Time Type Identification
SDK	Software Development Kit
STL	Standard Template Library
TCP/IP	Transmission Control Protocol / Internet Protocol
UI	User Interface

UML	Universal Modeling Language
WID	Wireless Internet Device
WLAN	Wireless Local Area Network

1 INTRODUCTION

1.1 Background

Software design has become an increasingly important task in software industry during the recent decade. The biggest need for good design is the growing size and amount of features in today's software. Object-oriented programming environments and platforms have helped to do the design with structuring and encapsulation. On the other hand, they have also increased the need for design specialists who understand the capabilities of object-orientation and also the need for tools, methods and models that enable powerful Object-Oriented Design *(OOD)*. Design patterns have been evolved from the use of such methods and models by those design specialists in their real life software development.

The most recent revolution in information technology is mobility. Computing devices are no longer bound to a specific location, they are mobile and move around in peoples backpacks and pockets. Such devices are no longer called computers or pocket computers. They are referred to as Wireless Information Devices *(WID)* or Personal Digital Assistants *(PDA)*. Those mobile devices combine the features of a phone, a calendar and a pocket computer. They have capabilities to handle all kinds of digital media available in the future broadband wireless networks. The most competitive platform for such devices is the Symbian Platform.

Software development for rapidly evolving field of PDA devices has to be swift, robust and reproducible. Most vertical applications have similar look and feel, functionalities and features outside and when peeking inside they often have very similar control structures, data flows and dynamic behavior. These facts must be considerer early in the design phase of a software process to enable rapid development of applications and to prevent work duplication.

1.2 Objectives and restrictions

The objective of this thesis was to study general design patterns, patterns that exist in EPOC and to research the use of design patterns in the EPOC environment. The first task was to identify and learn the most common design patterns and mechanisms behind them. Common design patterns are well defined and studied in the literature and in real life usage. The Gang-of-Four *(GOF)* presented their famous 23 design patterns already in 1995 /1/. Another approach of patterns was introduced in Pattern Oriented Software Architecture in 1996 /2/. This thesis will introduce a few of those patterns and study their applicability in the EPOC environment. The EPOC operating system has a large amount of frameworks and Application Programming Interfaces *(API)* that either are built using some patterns or force the application programmer to use a pattern. A few of those patterns are also presented and discussed in this thesis.

1.3 Structure of work

The thesis starts with an introduction to the notations, artifacts and terminology used later in the document. After terminology is a brief introduction to EPOC as an operating system in Chapter 3. Chapter 4 first presents some common OOD principles and then some principles specific to the EPOC operating system. Chapter 5 starts by introduction to design patterns, what they are, how they can be used and how they are categorized. Some of the most important EPOC patterns are presented in Chapter 6. Chapter 7 concretizes the use of some EPOC patterns in collaboration with common patterns and in the end briefly analyzes the design that was done. The example presented in chapter 7 is also discussed in /3/. The work done in the thesis is discussed and concluded in Chapter 8.

Chapters 3 to 6 are based on literature study and actual work done before and during the writing of thesis in Digia. Chapters 7 and 8 present the experience gained in the implementation phase of this thesis. The example application discussed in Chapter 7 is only presented in design level since the implementation details would have taken too much space and would not give any deeper insight in the scope of this thesis.

2 TERMINOLOGY

Object-oriented software field is full of different ways to express similar things. This chapter gives the reader an oversight about the techniques, terminology and notations used in following chapters. All terminologies used in this thesis derive from Digia operational models and processes.

2.1 Notations and language

The presentation of software in design level is most commonly done using diagrams. Architectural diagram specifies the component level structures and collaborations. Class diagram presents object classes and their relationships such as 'has', 'is-a', 'owns', etc. Modeling languages have specified unique ways to present those relationships and entities. Class diagram defines the static model of the design. The dynamic design model is presented with sequence diagrams. Sequence diagram presents object instances and the messages that are used for collaboration.

Architectural diagrams in this document are presented in block charts. Class diagrams are drawn using Unified Modeling Language *(UML)*. UML is the most common language in today's object-oriented modeling /4/. UML is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems /5/. UML is the proper successor to the object modeling languages of three previously leading object-oriented methods (Booch, OMT, and OOSE). Program code examples are presented in fixed font and a smaller size. Program code examples use Digia coding conventions /6/.

2.2 Diagrams

Design pattern structures in this document are presented as class diagrams. Class collaborations inside patterns are visualized using sequence diagrams. Architectural structures are presented in either component diagrams or plain block diagrams. Class diagrams use various UML artifacts to describe the class relations and the system

structure. Sequence diagrams contain nested method calls without signatures or return values for clarity.

2.3 Artifacts

UML diagrams have many different artifacts and all artifacts can be specialized to have unique meanings. This document uses simplified diagrams and only a few artifacts described below.

2.3.1 Mix-in

Mix-in is one type of class inheritance. Mix-in is an abstract base class that defines only an interface, which is implemented in derived class. Mix-in is used to add an interface to an existing class framework. Abstract class contains only the header and no implementation. Abstract base class cannot be instantiated. Mix-in is indicated with cursive font and <<abstract>> keyword in its methods. Figure 1 shows how mix-in can be visualized in UML.

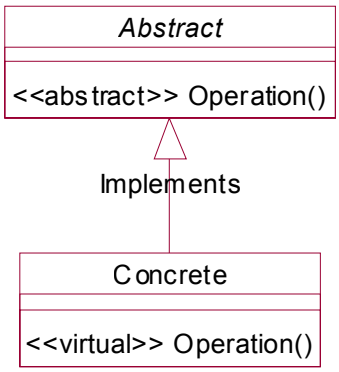

Figure 1. Abstract interface.

2.3.2 Specialization

Specialization is a typical case of class polymorphism. Base class defines an algorithm as well as an interface. Derived class specializes that behavior. Base class can specify default behavior and thus be instantiated or it can be an abstract class and leave the behavior specification to derivates. Specialization is indicated with a normal

inheritance symbol as shown in Figure 2. The base class has virtual methods marked with <<virtual>> -tag.

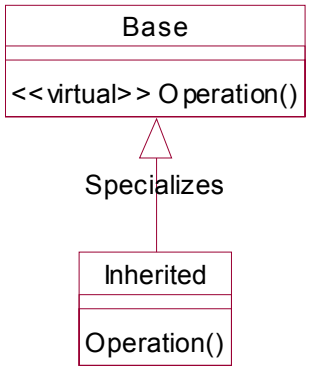

Figure 2. Specialization.

2.3.3 Composition

Composition forms a whole – part relationship between two classes. Whole – part relationship can be used for several purposes. The most common purpose is to encapsulate one to n relationship in data. Compound class can also define an algorithm, which is implemented using a sequence of atomic operations. Composition enables the run-time modification of the aggregates. Aggregate is indicated with a line having a diamond in the end of the owning class. When the relationship also contains delegation, there can be an arrow at the end of the line. Figure 3 shows the visualization of a normal type of composition.

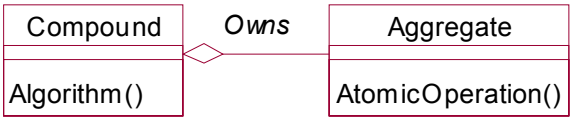

Figure 3. Composition.

2.3.4 Delegation

Delegation is used as class collaboration technique. A class delegates a task into a sub-unit, which can be a single class or a pattern of classes. The delegate can be changed in

run-time or be bind in compile time. Delegation is indicated with a solid arrow as shown in Figure 4.

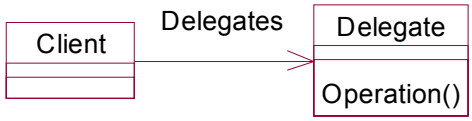

Figure 4. Delegation.

2.3.5 Instantiation

Instantiation arrow specifies the class that instantiates a specific class. Class instantiation is not indicated separately if it is obvious. Figure 5 shows the notation used for visualizing instantiation. Instantiation is indicated with a dashed arrow and can be explained more in detail with a note.

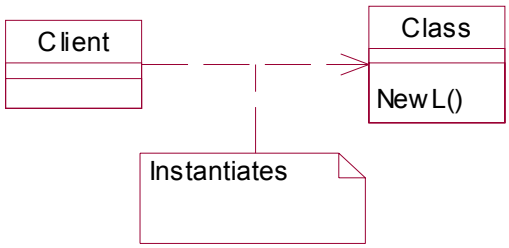

Figure 5. Instantiation.

2.3.6 Template

Template is a C++ specific technique to improve the reusability and versatility of source code. Template specifies a generic algorithm from which the compiler generates the type specific implementations. Template can be a whole class or just one method of a class or just one separate function. Class template notation is presented in Figure 6.

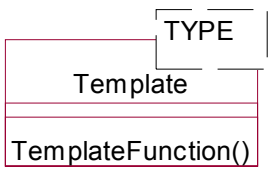

Figure 6. Class template.

3 EPOC

EPOC is an operating system developed by Symbian Ltd. This chapter briefly presents the EPOC operating system and some of its main features and characteristics of the environments it is mostly used in. The chapter is based on the Professional Symbian Programming book /7/ and other material released about EPOC in printed media and on the Internet /8/.

3.1 The Symbian Platform

The Symbian platform is a platform specially designed for wireless information devices often referred to as Personal Digital Assistant *(PDA)* or Wireless Information Device *(WID)*. The Symbian Platform is not just an operating system; it contains the middleware and applications as well. The Symbian platform is based on the early Psion operating systems /9/, SIBO and Protea, which were used in organizers and early pocket computers. The Symbian platform is intended to become an industry standard for the mobile community and has been licensed by all major mobile device manufacturers.

PDA devices are lightweight and have low memory consumption. In the Internet era all the PDA devices are communications oriented. People do not want to carry around separate devices for mobile communications, organizers, notebooks etc, so they are combined to form small hand-portable wireless information devices.

The Symbian platform defines three standard models of WIDs. They are defined as Device Family Reference Documents *(DFRD)*. A DFRD defines the general outlook of the device and the way it is used. The three currently defined DFRDs are called Pearl, Crystal and Quartz. Pearl devices are future smart phones; Crystal and Quartz devices are communicator style WIDs. Pearl reference design assumes that the device is a modern phone size lightweight device with a small color screen. Crystal devices have a real keyboard and can have a touch screen. Crystal devices are often foldable so that keyboard becomes visible after the device is opened as shown in Figure 7. The device

in Figure 7 is the Nokia 9210 Communicator. Nokia 9210 is a typical Crystal device without a touch screen. The original picture is from Nokia homepage /10/.

Figure 7. Nokia 9210, an example of Crystal DFRD.

Quartz devices are pen oriented, with a virtual keyboard and a few operational buttons. Quartz devices have also built-in handwriting recognition. Figure 8 shows an example of Quartz device. Original picture is from Digia SCC brochure at Digia homepage /11/.

Figure 8. Quartz Reference Device.

3.2 Special characteristics

EPOC is the operating system powering the Symbian platform. It is based on the early Psion organizer operating systems and is thus optimized for PDA and WID usage. WIDs are physically considerably small and they run on batteries. Size and weight set

quite strict limits for the hardware. The mass storage media like hard disks; CD- and DVD-ROMs cause most of the power consumption in a modern Personal Computer *(PC)*. They also require quite a lot of space. WID devices have no mass storage media so all users and applications data is in battery backed Random Access Memory *(RAM)*.

The device can be turned off in an eye-blink. When power is off the device does not consume batteries practically at all, only clock and some mandatory resources are processed. EPOC WID is practically on all the time. The switch from the sleep mode to active mode is fast and requires no loading of operating system or any other drivers. When the device is put to sleep mode no applications are closed, the CPU is just switched to halt mode and screen is put off. This means that the user can continue exactly from where he was left when he closed the device.

3.3 Hardware characteristics

The fact that the physical PDA devices are small and lightweight affects the design of their hardware system. EPOC is designed for 32 bit CPUs, running at rather low speeds compared to laptop computers. The power consumption of the CPU has to be small since the device is operating on rechargeable batteries most of the time. There is also backup battery to keep the user's documents safe when the main battery runs out. The device automatically turns itself off when batteries are low.

The Read Only Memory *(ROM)* holds the operating system and all the built in middleware and applications. System RAM is used for two purposes: for active programs and the kernel to store their data and as a disk space for user's documents. EPOC devices do not have any other storage media apart from the battery backed up RAM and flash memory cards. This sets quite strict limits to memory allocation and memory control overall. It is unaffordable that some application consumes all the free memory or leaks memory every time it is used.

EPOC devices have several input and output devices. Most devices have a keyboard and a touch screen for pen input. There is also an infrared port for communications

with other WID and IR –controllable devices like IR-printers. The device has also a serial port for communicating with a PC. PC connection is used for backup and synchronization purposes as well as for installing new programs to the EPOC device. Some EPOC devices are also integrated with a phone, either into the same device or with a separate device. Future EPOC releases will also have Wireless Local Area Network *(WLAN)* and Bluetooth support.

3.4 System structure

The Symbian Platform is completely object-oriented. Every EPOC framework and API is based on object-oriented solutions, which demands from developers that they use and are familiar with object-oriented methods and tools. That fact affects all software developed and ported for EPOC although the framework provides some help for old procedural style program porting. Version 6.0 EPOC releases have Software Development Kit *(SDK)* support for C++, Java and Organizer Programming Language *(OPL)*. C++ and Java are the SDKs for application development and OPL is mainly supported for Rapid Application Development *(RAD)* and for prototyping application ideas.

Figure 9 presents a slightly modified version of the Symbian platform system structure /7/.

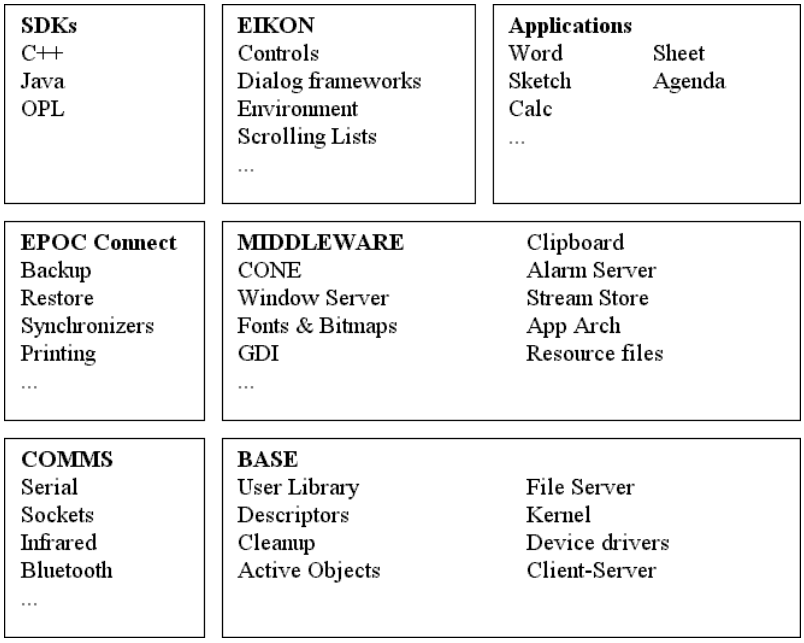

Figure 9. The Symbian Platform Structure.

Figure 9 shows the different groupings belonging to the Symbian Platform. The platform includes the built-in applications and SDKs as well as the operating system, EPOC. Base provides all EPOC programs with the fundamental APIs. Base includes the kernel, servers and their APIs. COMMS provides industry standard protocols for data communications, including dial-up networking, Transmission Control Protocol / Internet Protocol *(TCP/IP)* and infrared. On top of the base and COMMS are the EPOC Connect software and middleware such as streams, clipboard, etc. Middleware is a set of services, APIs and support utilities enabling powerful application development /12/. The topmost layer in Figure 9 has the SDKs for software development, UI frameworks and the base set of applications that belong to EPOC. The built-in application suite consists of typical office programs like Word, Sheet, Agenda, Contacts and Calc.

3.5 The EPOC operating system

EPOC is a layered operating system as shown in Figure 10. Only 20% of the code is different between the three DFRDs, which shows the effectiveness of layering. Other benefits of a layered architecture are discussed in Chapter 5.3.1. The topmost layer forms the UI architecture. The core of the operating system is the E32, which is the

same in all reference designs. The main layers above the E32 are the APIs for file server and the stream stores. Below the E32 are the device drivers and other machine dependent parts, the Hardware Abstraction Layer *(HAL)*. Even though the E32 layer is the lowest layer normal application programmer uses, the platform enables the use of device drivers straight without the E32. The layered architecture makes EPOC easy to port to new hardware since only the lowest layers, HAL and device drivers have to be modified or rewritten.

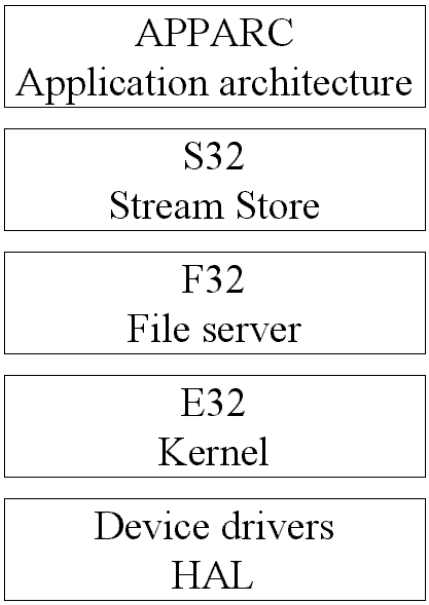

Figure 10. EPOC Architecture.

Since EPOC is an object-oriented operating system and PDA devices have limited resources, most of EPOC software development is done with C++ /13/. C++ is very comprehensive tool for software development. It has given many other aspects and paradigms to application programming than just inheritance and object orientation /14/. In a resource limited environment one has to be especially careful with the caveats and pitfalls of unnecessary inheritance, dynamic binding and exception handling /15/ although they are powerful tools when used correctly. EPOC has its own restrictions and principles for them as discussed in Chapter 4.3.

4 DESIGN PRINCIPLES

This section introduces the different principles behind every good software design. All design principles have the same goal, which is to avoid common pitfalls. The design principles are more like general-purpose guidelines. Most common principles are object-oriented and can be applied to all environments supporting object-orientation. EPOC principles are more strictly bound to the Symbian platform. Between those two types of principles is the safe area for the designer to do good software. Design principles are the first step towards design techniques and design patterns. The main tools to obtain the design principles in object-oriented environment are abstraction and encapsulation /16/. Those two mechanisms are behind all major principles.

4.1 Software design

Software design is a complex form of art. When an application evolves to design phase it is often not ready, features are not clearly specified and interfaces are fuzzy. Combined with a tight time schedule this often leads to a design that is impossible to reuse, complex and impossible to develop further.

There are four primary symptoms to indicate a poor design: rigidity, fragility, immobility and viscosity /17/.

Rigidity is the tendency for software to become difficult to change. Every change causes a cascade of changes in dependent classes and modules. Cascaded changes are often avoided adding unwanted dependencies, which lead to even bigger problems, usually fragility and viscosity.

Fragility is the tendency of the software to break up every time something is modified. This is very common when the design has illogical dependencies. Fragility makes software testing a nightmare. Each bug fix may break something and cause new bugs and all the tests must be run repeatedly.

Immobility is the inability to reuse software. Engineers often program modules that are partially generic and could be reused in other projects or even inside the same project. However, it often ends up in a situation where porting that module causes so much work that it's easier to rewrite it again. Adding new features to modules constantly often adds viscosity to the design.

Viscosity comes in two forms: viscosity of the design and viscosity of the environment. High viscosity in design means that when a change is required, preserving the design is so difficult that the engineer is more likely to do a hack and break the design. Improper tools and slow and inefficient environment cause high viscosity. High viscosity in environment causes engineers to optimize on compiling time instead of preserving and optimizing on design.

There are many steps in software process development to avoid the symptoms of bad design. Object-oriented software society is full of design principles and patterns, which help in the design and managing of the software. Those principles are mostly general and applicable to most programming environments. Most programming environments have also their own set of design and implementation principles.

4.2 Common object-oriented principles

All the common design principles presented in this chapter can be found in an article by Robert C. Martin: Design Patterns and Design Principles. /17/.

4.2.1 Open Closed Principle

The Open Closed Principle *(OCP)* is perhaps one of the most important Object – Oriented design principles /18/. The idea of the OCP is that in early phases of the analysis and design of the application the designers must decide, which parts of the system will be expanded later and which will stay solid for the whole lifecycle of the application. That decision enables the design team to start defining the abstract

interfaces inside the application and to start selecting design patterns to fulfill the abstractions and dependencies of the design.

OCP states that the design is extended by adding new code and classes rather than modifying the existing ones. New classes are inherited from existing frameworks and base classes. This requires that the frameworks have to be designed with a level of abstraction. Code that has been completed and tested is declared closed, so it will never be modified. Existing errors in code will be fixed, but new code will be put elsewhere. This makes the design open for expansion and adding features, but closed for modification.

Most design patterns use abstract interfaces and inheritance to keep the pattern easily expandable. When designing with design patterns the OCP will quite easily be fulfilled. The application programmer must also understand the patterns and the principles correctly in order not to break the principles in his application code.

4.2.2 Dependency Inversion Principle

The idea of Dependency Inversion Principle *(DIP)* is to depend on abstractions, not concretions /19/. This means that when you have dependency between two concrete classes A and B you should build an abstract interface in between then so that neither of the two classes have to depend strictly on each other. This way the design can later be expanded and the concrete class easily changed on each side without having to modify the interface in between them. Figure 11 shows the addition of an abstract interface in between the two dependent classes.

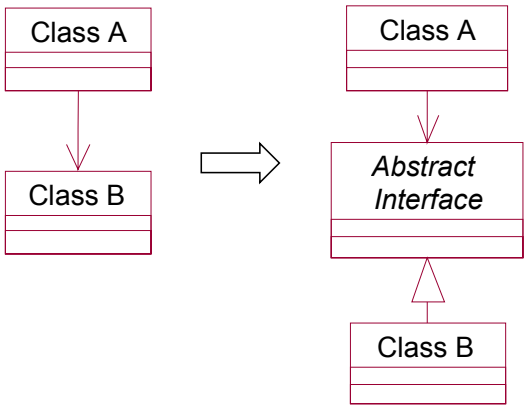

Figure 11. Dependency Inversion Principle.

The OCP sets the goal for an object-oriented design and the DIP is one of the most important methods to obtain it. However, there are a few places where you must depend on concretion instead of abstraction. The most common place is object creation. It is not possible to instantiate an abstract class. Creation of instances is possible throughout the application, so it might seem that you need to depend somewhat on concretion everywhere. This bottleneck can however be avoided by using the Factory pattern discussed in Chapter 5.4.1 or Abstract Factory /1/.

4.2.3 Interface Segregation Principle

Interface Segregation Principle *(ISP)* states that large interfaces should be divided into smaller ones /20/. Software designers often tend to combine several interfaces into single abstract classes to make it easier to modify the interfaces since they can be found in a single file. This packaging also leads to smaller class hierarchies and might falsely seem more efficient. However, it is often indicated that combining interfaces makes it impossible to vary or reuse them independently. Smaller interfaces let us have smaller granularity on concrete class interfaces and allow us to define clearer roles for entities inside the class structure. Clients should not be forced to depend upon interfaces that they do not use.

4.2.4 Acyclic Dependency Principle

The Acyclic Dependency Principle *(ADP)* states that dependencies between software entities must not form cycles /21/. Simplified this means that if class A depends on class B, class B must not be dependent on class A or it's base classes. Most design tools and C++ compilers make it possible to program dual dependencies using thin declarations. Dual dependencies lead to a situation where modifying one class forces us to modify the other also. To avoid cascaded changes it is better to use single declarations and to go back to the design board when a cyclic dependency occurs. There also exists design patterns to prevent acyclic dependencies /22/ /23/.

In iterative software development process, new features are added to the design during the whole development time. This often leads to tight and unwanted dependencies inside modules and between them. This is because there is never enough time for proper design in the beginning of the process and new features are being added all the time. New features require modifications to the class hierarchy. At some point of the development, the hierarchy becomes impossible to change and the designer is forced to make some "dirty" solutions, which end up adding dependencies between logically unrelated classes. To avoid this the design needs to be refactored continuously /24/. Using design patterns and following the OCP and DIP greatly decreases the need for refactoring, since the model is open at the places where the new features fit into the class hierarchy.

4.3 EPOC principles

EPOC is a very restricted environment from a developer's point of view. The restrictions result mostly from the special characteristics discussed earlier in chapter 3.2. The EPOC principles presented here are from the Symbian Quartz 6.0 C++ SDK /25/, the Professional Symbian Programming book /7/ and from the Symbian Developers Network Homepage /26/.

4.3.1 Memory allocation

Each thread in an EPOC application has a limited standard stack space of 8Kb, which should be carefully managed. The stack is allocated during the thread creation so the programmer can always trust it to exist. To prevent stack overflow developers must avoid copy-by-value, except for basic types. Because of small stack size, programmers must minimize the lifetime of automatic variables by appropriately defining their scope.

All large classes and arrays must be created to the heap rather than the stack. This adds an extra possibility for programs to leak memory. Since the heap can run out at any point of the program execution, all heap-allocated objects must be carefully constructed and it can never be assumed that the construction fully succeeds.

4.3.2 Function overloads

If a function definition has default arguments, and if that function is often called with the caller assuming the default arguments, the programmer should consider providing an overloaded function that doesn't have the additional arguments. This is because every time the compiler supplies a default parameter, it generates additional code where the function is called.
So instead of:

```
Function( int aValue = 0, int aValue2 = 0 );
```

Should be defined as:

```
Function( void ); // calls Function( 0, 0 );
Function( int aValue ); // calls Function( aValue, 0 );
Function( int aValue, int aValue2 );
```

4.3.3 Object code

Avoidance of object code duplication is a difficult issue in a C++ implementation. Especially templates can be difficult to manage. A template really defines a whole family of classes /15/. Each member of the family that is instantiated requires its own object code.

In EPOC operating system, object code duplication must be avoided at all costs. This is especially important for programs that will reside in the devices ROM memory. Object code duplication can be avoided by using effective algorithms, avoiding copy-pasting code around and using the thin template idiom described in Chapter 6.5.2.

4.3.4 Inline functions

Inline functions are intended to speed up code by avoiding the expense of a function call, but retain its modularity by disguising operations as functions. Before using them, two issues should be checked:

- Code compactness: limited memory resources may mean that the speed cost of a function call is preferable to large bodies of inline code.
- Binary compatibility: changing the implementation of an inline function can break binary compatibility. This is important if your code is going to be used by other developers.

The most common cases where inline functions are acceptable are:

- Get and set methods for one- or two-machine word quantities: for example,
  `inline ConEnv() const { return iConEnv; };`
- Trivial constructors for T classes:
  `inline TPoint::TPoint(TInt aX, TInt aY) { iX=aX; iY=aY; };`
- Certain other operators and functions whose definition is not subject to change and purpose is to map one operation onto another, for example:
  `template <class T> inline T Min(T aLeft,T aRight)`

```
{ return(aLeft<aRight ? aLeft : aRight); }
```

4.3.5 Typecasting

Typecasting is used when an instance or a reference of a specific type has to be converted into another type used in this context. Typecasts should be used with caution, as in other operating systems. If a cast seems to be needed, it should be checked that this does not reflect a design weakness. Typecasts are the most common cause for unfound bugs that crash programs after they have been released /27/.

EPOC provides its own macros to encapsulate the C++ cast operators. These should be used in preference to using the C++ operators explicitly. Programs for hardware running EPOC are compiled mostly using the Gnu C Compiler *(GCC)*. GCC currently has a poor support for typecasts so in GCC compiles the macros just replace the C++ casts with plain C-style casts. Current EPOC releases do not support Real Time Type Identification *(RTTI)* /13/. For this reason the more sophisticated C++ dynamic_cast operator can not be used. This also disables the use of some design patterns in EPOC that are based on RTTI /22/.

4.3.6 Resource acquisition

EPOC devices have a limited set of resources and limited performance. It is therefore crucial that application developers free the resources they use after they are not needed. Open handles and references cost memory and slow down performance. The operating system frees some of the resources automatically, but there is always some time gap. Some resources are not freed until all handles for it have been closed.

For example a thread, even when it has been killed, will not be removed until all open handles to it have been released. This is because the removal also removes the reason why it terminated as well as all other information. Therefore, when application

programmer opens handles to threads he must remember to close them too or the system will start degrading because of "ghost threads".

4.3.7 Coding Conventions

EPOC has strict coding conventions to ensure software quality and to improve the readability and understandability /25/ /6/. Class names begin with a letter indicating its type. T-classes are flat small classes that can be allocated from the stack. Flat means that their size is constant.

C-classes are compound classes that have two-phase construction because they have to allocate some memory for their resources. C-classes must all be derived from *CBase* class and allocated dynamically from the heap. A C-class destructor must never assume a full construction. Temporarily created C-classes must be pushed into cleanup-stack if there is a possibility that the program might exit before the correct deletion of the class.

```
CThing* temp = CThing::NewL(); // NewL handles construction
CleanupStack::Push( temp );
SomethingL(…);
CleanupStack::PopAndDestroy(); // pops and deletes the temp
```

M-classes describe only an interface so they can only be used as a reference. Mix-in classes are not allowed to have any implementation. R-classes are references to EPOC resources such as fileserver session, threads, handles, etc. R-classes can be allocated from stack and therefore need no cleanup handling. Most R-classes require a connection to be opened before operating with the resource and closed after the use.

```
RFs fsession;           // create a handle to fileserver
fsession.Connect();     // open connection, start session
…           // use fileserver session
fsession.Close();       // close connection, free resources
```

Instead of C++ style exception handling, EPOC has a TRAP - Leave exception handling /26/ /25/. When there happens an error that requires attention, the program calls `User::Leave()` method. When a leave occurs the execution of the program returns to the last TRAP, after which the developer can program the exception handler. Methods and functions that may cause a leave must have an L as their name postfix.

```
void DoSomeL( void )
      {
      User::Leave( 0 );
      }
TRAPD( error, DoSomeL() );
If( error )
      // handle error situation
```

The L postfix tells the user of the class that the method does something that might fail and requires error handling. C postfix at the end of a method tells that the method leaves something to the cleanup stack, and the user has to remove it from there later. D postfix after the method name informs that the method deletes the object that it gets as a parameter.

```
CThing temp = CThing::NewLC(); // temp is now created and put to
                               // cleanup stack
…     // use temp
ProcessLD( temp );  // this may leave so keep temp in cleanup stack
CleanupStack::Pop(); // remove temp from stack since ProcessLD
                              succeeded
```

Coding conventions are a very informative way of telling the user about the method and class behaviour and resource usage inside it. They are also a very effective way of checking possible errors in a code review. Every EPOC programmer must be aware of all the conventions because the frameworks use them everywhere. A misunderstood or misused convention leads into unpredicted behaviour, unstable programs and unusable interfaces.

5 DESIGN PATTERNS

This chapter describes what design patterns are and how they can be used and categorized. Different patterns types will be introduced with a few illustrative examples. Architectural patterns will be visualized with component diagrams. Design patterns will be presented using a formal notation with UML class diagrams and sequence diagrams. Most of the design patterns presented here are among the 23 “Gang of Four” patterns /1/.

5.1 Definition

Design patterns are “descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context” /2, page 3/. Design pattern tend to be simple, yet effective solutions to problems that arise repeatedly in object-oriented modeling of software. Design patterns can also be seen as higher-level reusable building blocks /3/. Design patterns can be general and usable in various problem domains or they can have a very strict problem domain where they best fit in. A design pattern rarely solves a design problem on it’s own. Patterns often need to be modified and specialized to properly fit the software environment and the problem domain /28/. Patterns also rarely exist alone. Most software architectures /29/ contain pattern systems having patterns co-working and even many patterns merged as bigger compound patterns. Design patterns are a part of a whole family of software patterns containing Process Patterns /30/, Analysis Patterns /31/, Architectural Patterns /29/ /2/ /12/, Design Patterns and much more.

5.2 Categorization

Design patterns can be categorized in many different ways. The categorizations rarely exclude each others. They often just present the patterns in another perspective. Figure 12 shows three different categorization methods combined into a single table. Patterns presented in bold text in the table are also discussed in this thesis. Abstract Factory, Template Method and Iterator can be found in Design Patterns /1/. Another aspect for

iterator can also be found in the C++ Standard Template Library *(STL)* /32/. The patterns in Figure 12 are both general and EPOC patterns.

The most common categorization method is the GOF /1/ method, which categorizes design patterns into three categories by their task in the application. The GOF categorization is in the horizontal axis of Figure 12. Creational patterns abstract the object instantiation process, Structural patterns define ways how larger structures are formed from classes and objects and Behavioral patterns implement algorithms and assign responsibilities and tasks between classes and objects. Another categorization method is to divide patterns to static and dynamic structures /33/. The idea behind static and dynamic division is that static and dynamic modeling of software often happens in separate phases of the software development.

		Architectural Patterns	Design Patterns	Design Idioms
Creational	Static		**Factory Method**	**Two-phase construction**
	Dynamic	Abstract Factory	**Singleton**	**Cleanup stack**
Structural	Static	**Microkernel**	**Class Adapter**	**Thin template**
	Dynamic	**Model–View–Controller**	**Object Adapter**	
Behavioral	Static		Template Method	**Exception handling**
	Dynamic	**Active objects**	**Observer**	Iterator

Figure 12. Pattern categorization.

Another way to categorize patterns is by their scale and abstraction /2/ /12/. The scale categories are on the top of Figure 12. The highest level of patterns in this categorization is the architectural patterns. Their task is to support the refinement and modeling of subsystems and components. The mediate level patterns are the design patterns help to implement particular design aspects and solve domain specific design problems. The lowest level of patterns is idioms. Idioms help to implement the particular design aspects and higher-level patterns. The vertical division on patterns presented in Figure 12 is not as strict and clear in real life as it may seem. Many architectural patterns can be considered design patterns and vice versa. The horizontal

division is clearer even though patterns from different categories can be used to solve similar problems.

5.3 Architectural patterns

Architectural patterns describe the large-scale structures of the software. Architectural patterns provide a set of subsystems, specify their responsibilities and include rules and guidelines for organizing the relationships between them. The architectural pattern presented here can all be found in Pattern Oriented Software Architecture /2/.

Software developer very seldom has the privilege to select the architecture for the design; it is defined by the business models and the platform that is used. Design tools and other environmental factors can also specify the architecture or some of its restrictions.

The meaning of architectural patterns in the design pattern context is primarily to understand the higher-level abstractions and the way that applications are meant to be built within a specific environment. The architectural patterns often define the lower level design patterns or lead to the selection of them. Architectural patterns are the basis for frameworks in the software development environment and frameworks define the individual patterns to be used in a specific task on an application.

5.3.1 Layers

Layers architectural pattern describes a design that is divided into several layers on top of each other. Each layer has its own tasks in the whole system and interfaces to the neighboring layers.

Most modern interactive operating systems are based on layers. The highest layer is the user's interface to the system and the lowest level is the kernel and the device drivers. In between there are several layers defined for specific tasks like event handling,

drawing, file managing, etc. The application programmer chooses the level of detail he wants to deal with and designs the application using the services provided in that layer. Most layered systems require the application to interact with several layers and the more freedom the programmer needs the lower layers he must use.

Data communication protocols like TCP/IP are usually designed as a layered system. Operating systems also use layering to improve modularity.

Structure

The layers architectural pattern can be implemented in several ways. The best-known solution is the top-down approach where the client issues a request from layer N. If the layer N can't carry out the request, it passes it to the N-1 layer. Then the N-1 layer tries to serve the request and calls N-2 and so forth. When the correct layer is found to serve the request, the result is carried back to the client through the layers.

Another solution is that two stacks of N layers are communicating with each other. This is a well-known scenario from communication protocols, where the stacks are known as protocol stacks. The client request moves down from layer N to layer one where it is sent to the other stack and upwards on the corresponding layer N there.

Consequences

Individual layers can be reused if they are well defined and well documented. Dependencies in layers are kept local inside one layer. A layered architecture is very modular and can be tested individually on each layer. Layered architectures also have good portability and scalability. Typical upgrades and modifications of the architecture require changes only in individual layers.

The biggest disadvantage of layered architecture is that when a big architectural requirement changes the modification cascades through all layers, it thus creates enormous amount of work. However, big changes happen very rarely and they can be

avoided with careful planning and design. A layered architecture has more interfaces and is more complex than a monolith system, and therefore suffers somewhat in efficiency.

5.3.2 Model-View-Controller

Model-View-Controller *(MVC)* pattern is perhaps the most varied architectural pattern. It is the basis for most interactive user interfaces. Model contains the data and the program logic, view describes how it will be represented to the user and controller handles user inputs and commands. MVC pattern enables quite fluent porting of existing systems to new environments. Most of the modern operating systems support MVC patterns in framework level so the only thing needed for implementation is a group of adapters to overcome the platform specific implementation problems. The MVC pattern divides an interactive application into three components /2/. The structure and relations between the components in the MVC pattern is shown in Figure 13.

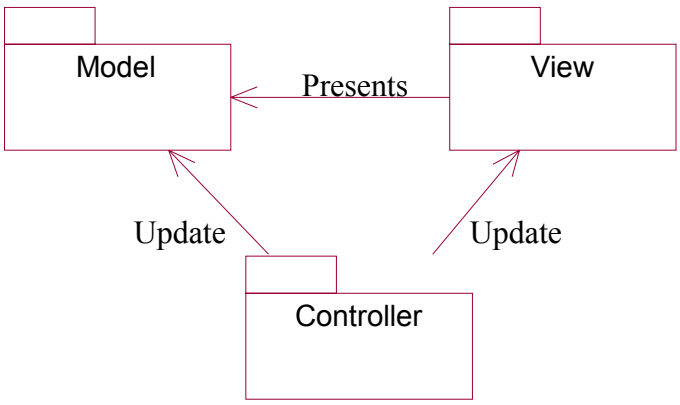

Figure 13. Model - View – Controller architectural pattern.

The model is an independent entity. The model encapsulates the application data. The view uses the model. Its task is to represent the data by drawing it to the screen or some other viewing device. The controller coordinates the model and view updates.

Consequences

Model-view-controller architecture separates the model from its presentations and therefore provides the possibility for multiple views of the same model. MVC pattern enables easy changing and customization of the 'look and feel' since the user interface is separated from general application structure. The separation also makes it easy to port the application to a new platform. Only the user-interface components need rewriting. If the view part of the application is built and designed properly, the porting only requires a few adapters to be written. Adapter design pattern is described in Chapter 5.5.1.

5.3.3 Microkernel

The microkernel architectural pattern introduces a software system, which can adapt to changing system requirements and environments. Microkernel separates minimal functional core from extended functionalities and customer-specific parts. A microkernel design pattern greatly improves the portability of the platform. Modularity of the microkernel makes upgrading and customizing the platform easier than monolith architectures.

Structure

Figure 14 presents the five main modules of the microkernel architecture.

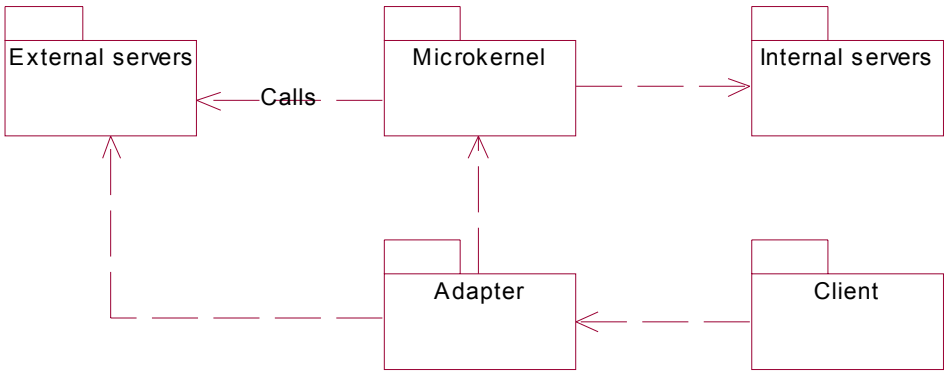

Figure 14. Microkernel architectural pattern.

The participating components are:

- Microkernel
- Internal servers
- External servers
- Adapters
- Clients

Microkernel component implements atomic services that are needed for all applications throughout the system. The microkernel has the functionality for inter-process communications. Microkernel also maintains system wide resources and controls and coordinates the access to them. Core functionalities that cannot be implemented within the kernel without unnecessarily increasing its size and complexity are separated into internal servers.

Internal servers extend the functionalities of the microkernel. Internal servers can for example handle graphics and storage media. Internal servers can have their own processes or they can be shared Dynamic Link Libraries *(DLL)* loaded inside the kernel.

External servers implement their own view of the underlying microkernel. External servers use the services of the microkernel and internal servers to provide their own services for the clients. External servers run in their own processes.

A client is the application that is associated with exactly one external server. The client uses the communication services provided by the microkernel to communicate with the server it is associated with. The client provides the application-programming interface *(API)* for using the external server.

The role of the adapter is to provide a transparent interface for clients to communicate with external servers. Adapter hides the system dependencies such as communication facilities from the client. Adapter thus improves the scalability and changeability of the system. The adapter enables the servers and clients to be distributed over a network

Consequences

The microkernel pattern improves the system portability since in most cases you only need to modify hardware dependent parts when migrating the architecture to a new hardware. The external server and client code can remain the same.

The biggest advantages of the microkernel are the flexibility and extensibility. If a new device is added to the system, all that has to be done is to write a new server for it. No modification to existing client, server or kernel code is required.

Compared to monolith architectures the microkernel system requires much more inter-process communication inside one application execution because of the calls to internal and external servers. If the system is not optimized for communication and context-switch the slow execution speed of applications is the price we have to pay for flexibility and extensibility.

The design and implementation of the microkernel -based system is far more complex than of a monolith system.

5.4 Creational patterns

Creational patterns encapsulate knowledge about the concrete objects the system uses and the way in which they are instantiated. The rest of the systems use the objects via the abstract interfaces and have no special knowledge about the concretion. Depending on abstraction rather that concretion is also the goal of the DIP principle so using creational design patterns greatly improves the flexibility of the design.

Creational patterns are closely interrelated as they all deal with object instantiation. The biggest difference between different creational patterns is the ownership of the created instances.

5.4.1 Factory Method

Factory method defines an interface for creating an object, but letting subclasses decide which concrete classes to instantiate. Factory method works as an abstract interface between the client and the concrete objects. A variation of factory method is the parameterized factory method, where the parameter defines which concrete class to create. This is really useful when the objects to be created are defined in the run-time.

Structure

Figure 15 shows the structure of generic factory method pattern. Client depends on the abstract creator and product classes. Creator and concrete creator classes may also implement template method design pattern /1/.

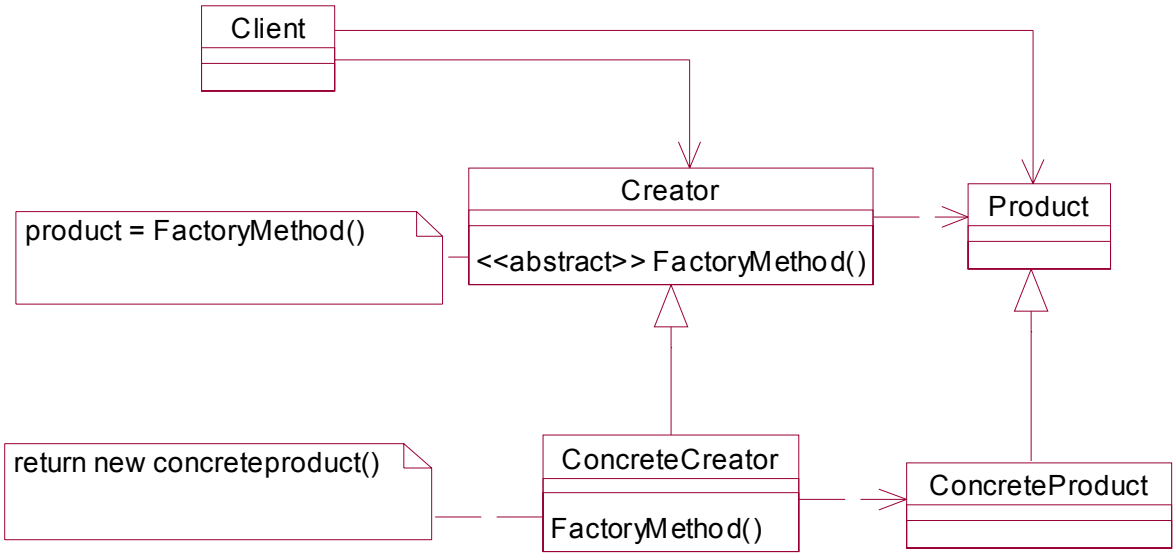

Figure 15. Factory method design pattern.

Participants

- Product
  - The abstract interface of objects that the factory method creates.
- Concrete product
  - The concrete object that the concrete factory instantiates.
- Creator
  - The abstract interface for creating the products. Introduces the factory method prototype.
  - May contain a default implementation of the object creation.
- Concrete creator
  - Instantiates the concrete products. Implements the factory method, which returns the concrete product.

Collaborations

Creator depends on its subclasses to define the factory method so that it returns the appropriate concrete product. The client uses the abstract creator class to get product

instances. Figure 16 shows the sequence where client creates a product through factory method.

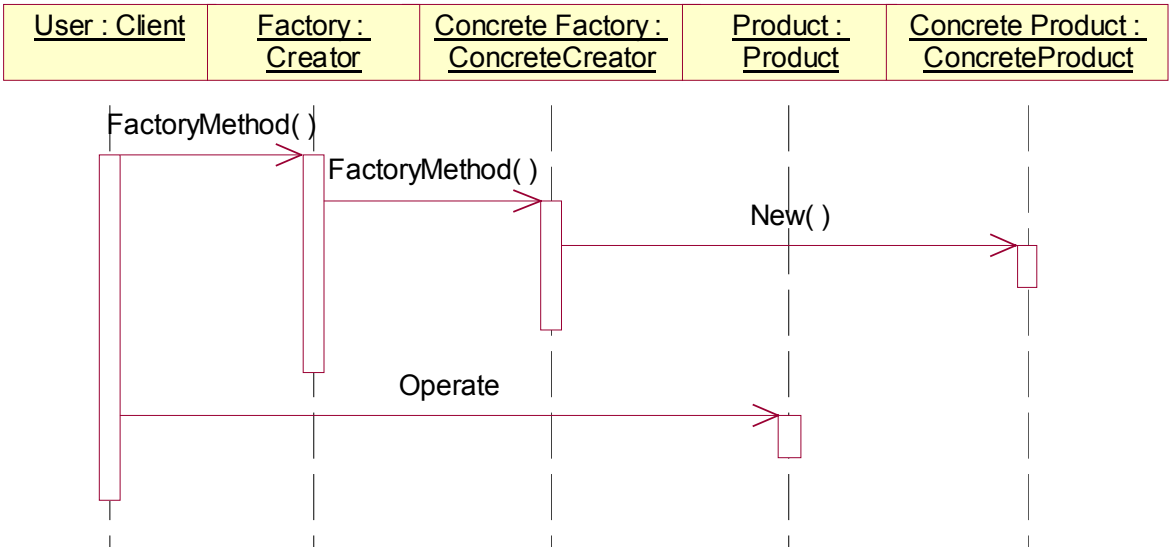

Figure 16. Factory Method Sequence Diagram.

Consequences

Factory method eliminates the need to bind application specific classes everywhere into application code. Therefore, the decision of the actual concrete classes can change later in the application design or even in runtime. The design is also easier to divide into smaller parts when the implementer of the factory can work at the same time as the users of the factory since they both have abstract interfaces to rely on.

Factory method provides a very clever way to centralize object creation. Factory method makes it possible to design systems that are very reusable from the core implementation. By changing the concrete factory class, the same algorithms for different problem domains can be used. Factory method is very often used with other patterns to form a pattern system.

Following the OCP and DIP design principles the software should be designed relying on abstraction and keeping the code open for expansion and closed from modification. With the aid of factory method, it is easy to obtain both of those goals. The design can

be expanded by adding new concrete products and modifying only the factory. The users of the factory need no modification since they rely on the abstract product interface. Even the modification of an existing factory can be avoided by inheriting a new factory and only expanding the factory method with support for new concrete products.

5.4.2 Singleton

Singleton is probably the most discussed, argued and varied design pattern. The idea of the singleton design pattern is to guarantee that an object has one and only one instance. Usually the instance is also made globally accessible.

Singleton pattern can also be used to guarantee that an object has a limited amount of instances. This feature is obtained using the reference counting idiom. The implementation of singleton pattern varies in different programming languages and environments and therefore singleton is often considered more an idiom than a design pattern.

Structure

Typical structure of singleton design pattern is presented in Figure 17. Client uses singleton as a normal instance of the actual class. Singleton is therefore transparent for the users.

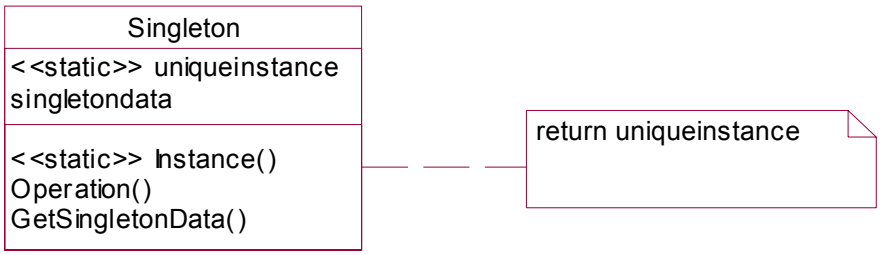

Figure 17. Singleton design pattern.

Participants

- Singleton
    - Defines the interface for clients to globally access the unique instance of singleton.
    - May be responsible for creating and destroying its own unique instance.

Collaborations

Clients can access the singleton instance only through singletons interface. Singleton can also control the access to the instance. Singleton can be seen as a proxy for the real data or object.

Implementation

The implementation of the singleton pattern requires either a global registry, which keeps track on the allocated singletons, or a class specific static variable that points to the sole instance. In most environments this does not cause any problems, but in EPOC writeable static data is forbidden in any DLLs including applications due to architectural reasons discussed more deeply in Chapter 6.2. This makes it difficult to implement an application type independent singleton in EPOC and the use of singleton should therefore be avoided in general EPOC programs.

Consequences

Singleton provides controlled access to the sole instance of the desired resource. Singleton provides global access to a single, perhaps limited resource. Singleton often has to take care of allocating the instance when it is referenced for the first time and to deallocate it when the program is finished.

5.5 Structural Patterns

Structural patterns compose larger structures from classes and objects. Structural class patterns use inheritance to compose interfaces or implementations. Structural object patterns describe ways to compose objects in order to realize new functionality.

5.5.1 Adapter

Adapter is one of the most used design patterns in software design. However, most adapters in existing systems are added to the system in implementation phase rather than in design time. The reason for this is that the application often has to be adapted to changing requirements and therefore adapters have to be added to keep the structure of the original design. Another name for adapter design pattern is wrapper as it wraps existing functionality to work in new environments. Adapter is a very essential pattern when designing portable applications and porting existing applications.

Structure

Adapter can be implemented in two ways: either as a class adapter or as an object adapter. The class diagram of a class adapter is shown in Figure 18. The class adapter uses inheritance to obtain adaptee's behavior.

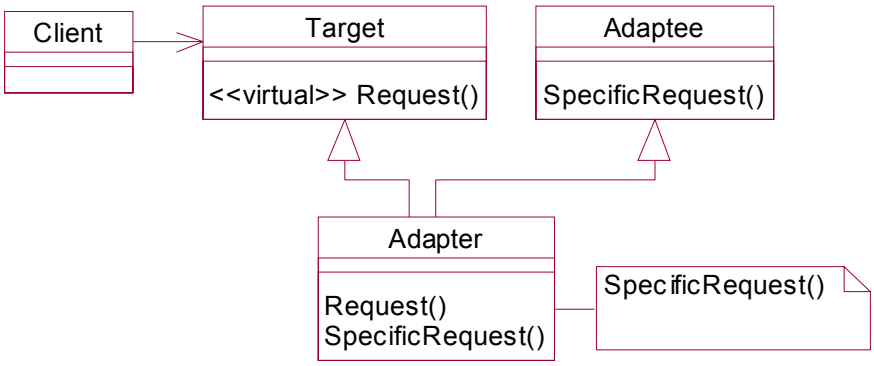

Figure 18. Class adapter design pattern.

Object adapter is presented in Figure 19. Object adapter uses aggregation to enable message transforming into adaptee's interface.

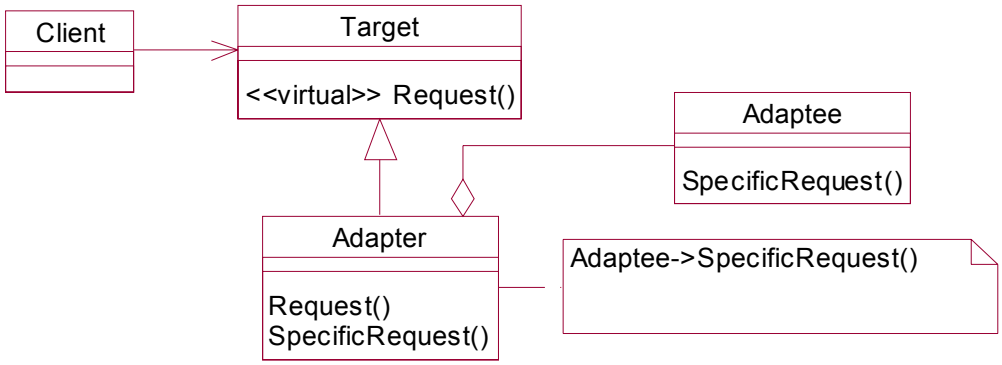

Figure 19. Object adapter design pattern.

Participants

- Target
  - Defines the interface for this domain.
- Client
  - Collaborates with objects using the target interface.
- Adoptee
  - Defines an existing interface that needs adapting.
- Adapter
  - Adapts the existing adaptee's interface to the new domain.

Collaborations

Client uses the adapter via the interface that target defines. Adapter translates the client calls to corresponding adaptee's methods. Adapter is transparent for the client.

Consequences

Class adapter adapts a concrete adoptee class to target since the adapter is directly inherited from the adoptee via private inheritance. This prevents the use of adaptee's subclasses as adaptee without rewriting the adapter.

Class adapter also enables adapter to override adaptee's behavior since adapter is a subclass of adoptee. Class adapter is one concrete object and requires no pointers or references to objects elsewhere. Class adapter can also be a two-way adapter when it inherits from two adaptees. Such adapters provide transparency and can be used in both adaptees' environments.

Object adapter enables single adapter class to be used with all adoptee type objects. The adapter can also add functionality to all adaptees at once. However, the object adapter has no way to override the adaptee's behavior.

5.5.2 State

The intent of the state pattern is to provide an entity the possibility to alter its behavior when its internal state changes /1/. State pattern also lets us vary the state based behavior by modifying the state transitions and reuse the behavior separately.

Structure

Figure 20 shows the structure of state pattern. The actual amount of concrete state subclasses may vary.

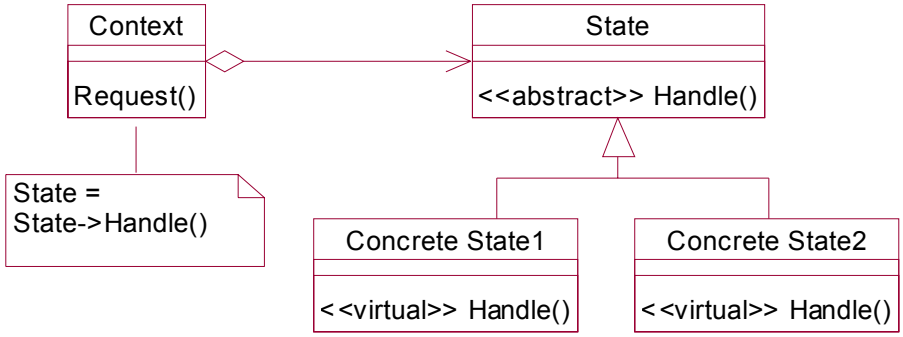

Figure 20. State design pattern.

The state pattern describes a context, a state machine, which defines the entity that has state based behavior. It also provides a uniform interface for the states. The interface is abstract in most cases but can also contain some state shared behavior or data. The transition from one state to another is usually defined inside the concrete states, but it can also be defined inside the state machine. Declaring it inside the states decreases the need to modify the context when state transitions are modified or new states are added.

Participants

- Context
  - Defines the state machine.
- State
  - Defines a uniform interface for the concrete states.
- Concrete state
  - Implement the state based behavior.

Collaborations

Figure 21 shows the sequence diagram of a state change in state design pattern.

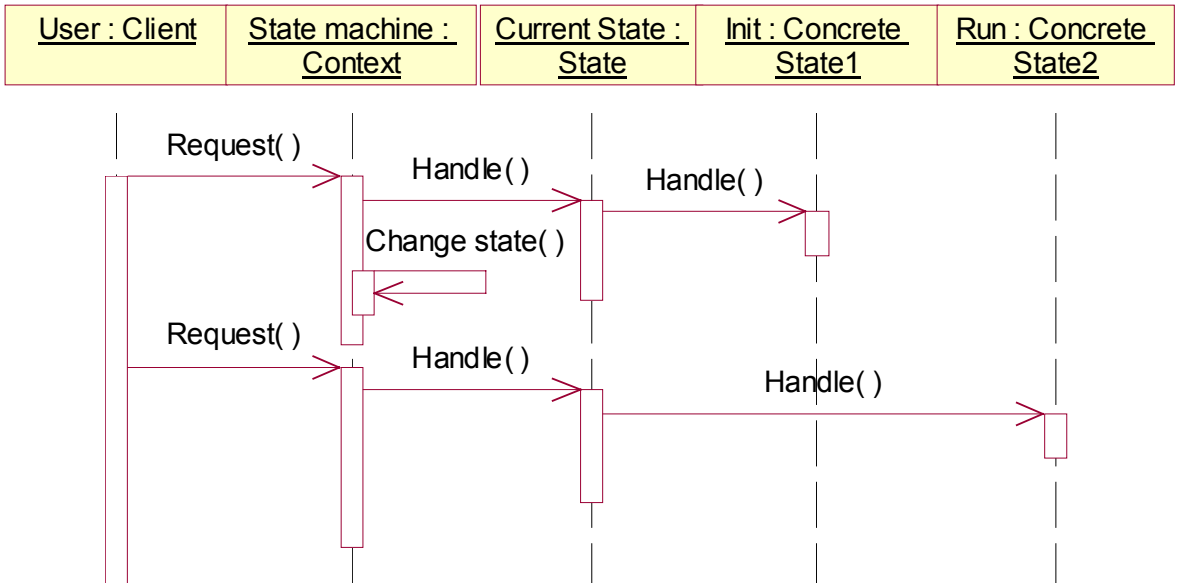

Figure 21. State Sequence Diagram.

The state change is invisible for the user; it has the same interface even though the inner functionality of the state machine changes.

Consequences

State design pattern enables us to design very flexible and expandable solutions. The state transition can be defined by the state subclasses or by the state machine itself. The difficulty in implementing the state pattern arises when all the states use the same resources. The resources are often defined in the state machine and the subclasses must get a reference to the owning class, which is not a very elegant solution.

5.6 Behavioral patterns

Behavioral patterns describe object communication patterns, algorithms and their implementation and control flow handling.

5.6.1 Observer

Observer defines a one-to-many dependency between collaborating objects. Observer enables the partitioning of a system into observers that react when their subjects change state. Observer is used in many event-based systems to separate the event source from the event monitors.

Structure

Figure 22 shows the structure of observer design pattern. The concrete observers attach to the subject so that the subject knows which observers to notify when its state changes.

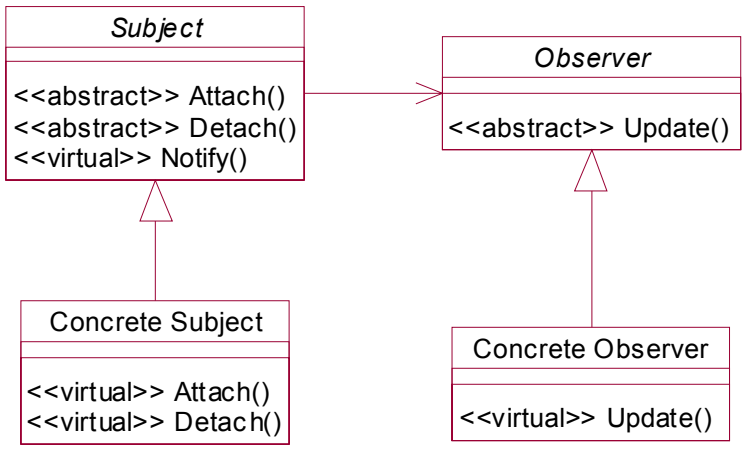

Figure 22. Observer design pattern.

Participants

- Subject
  - Defines the interface for the observers to attach and detach themselves for a subject.
- Concrete subject
  - Implements the list of interested observers and the notification logic.

- Observer
    - Defines the abstract update method that the subject calls when it changes state.
- Concrete observer
    - Implements the concrete update method for the subject to call.

Collaborations

Observers register themselves to the subjects they are interested in. A subject may have multiple observers and an observer may listen to several subjects. The update method may contain a flag indicating which subject changed state in case of many-to-many relationship.

Consequences

The problem with standard observer pattern is that type information gets lost if the subject or observer hierarchies are derived /28/. A single observer does not know which subject sub-class changed state. There are however several special design patterns that solve the problem /22/. Nevertheless, some of them are not applicable for EPOC environment because they require RTTI to work.

6 EPOC PATTERNS

EPOC is a fully object oriented operating system and is designed with great care on modularity and flexibility. This chapter introduces a few of those architectural and design patterns and idioms. Some of the patterns presented here were used in the implementation phase of this thesis and they are also discussed in Chapter 7.

6.1 System architecture

EPOC is built to suit in many different types of small devices. This suitability requires modularity and flexibility. Those characteristics are easily achieved with layering. EPOC system architecture is a typical example of layers architectural pattern as shown in Figure 23. Figure 23 is a slightly modified version of the original diagram presented in Professional Symbian Programming /7/.

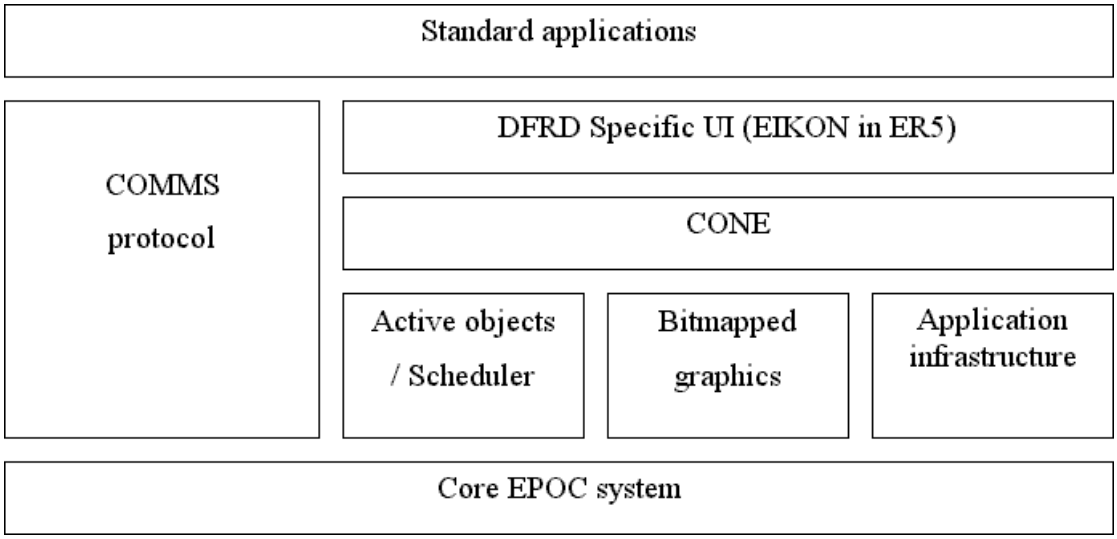

Figure 23. EPOC System Architecture.

The layered architecture makes EPOC a very modular operating system. Core EPOC system includes the basic types like descriptors, exception and leave mechanism, processes, thread handling, files, streams, stores, etc. They are the same for all different EPOC implementations. Active objects and scheduler are the basis for asynchronous

processing. Bitmapped graphics component contains generic bitmap handling and the ability to draw or print into real devices. Application infrastructure handles application launching, file associations and other supporting actions. COMMS protocol handles the communications and support for protocols. CONE is the graphical control structure of EPOC providing the abstract controls and the framework. UI level specifies the look and feel of the user interface and provides the concrete controls to implement it. In EPOC release 5 the UI level is called EIKON.

Applications are built on top of these layers using the services of several layers. Simple applications only need to deal with the topmost layers, mostly EIKON and CONE. Complex applications like client-server applications need to get deep inside the core layer and to the kernel services.

The EPOC graphics architecture is also a layered structure. Figure 24 shows the layering of EPOC release 5 graphics system.

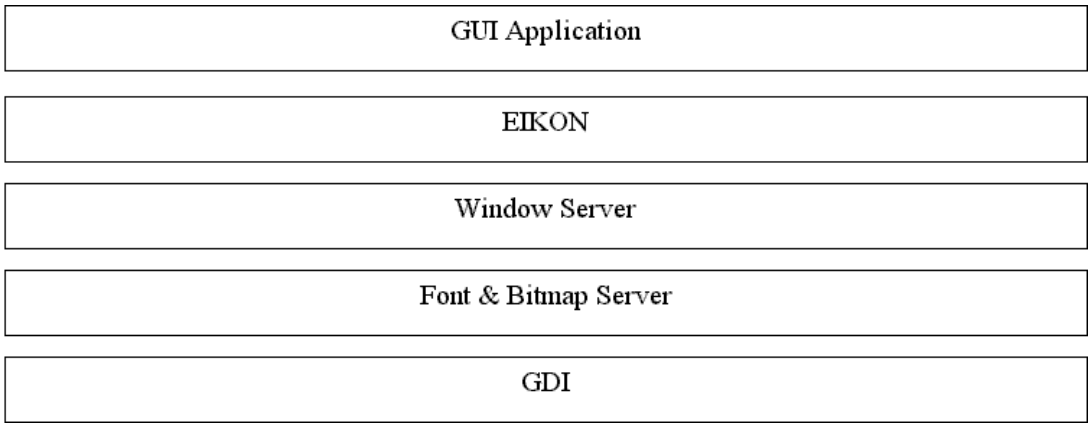

Figure 24. EPOC Graphics Architecture.

At the bottom is the Graphics Device Interface (GDI), which defines the drawing primitives and provides device independent drawing. Font and bitmap server provides the system with fonts and bitmap handling functions. Window server manages screen, pointer and keyboard on behalf of all GUI applications within the system. EIKON provides reusable controls, menus, buttons, dialogs, etc.

A simple graphic method call in the application level derives through the layers into several calls in the GDI level. However, the user has the freedom of choosing the level of detail and abstraction by selecting the appropriate layers to use to get the most flexible and efficient solution for his graphical needs.

EPOC Kernel is an example of the microkernel architectural pattern. Figure 25 shows the fact that kernel servers are run in user mode. The kernel server runs privileged and is the highest priority thread in the system. The vertical lines in the Figure 25 show different threads and the dashed line indicates the privilege boundary.

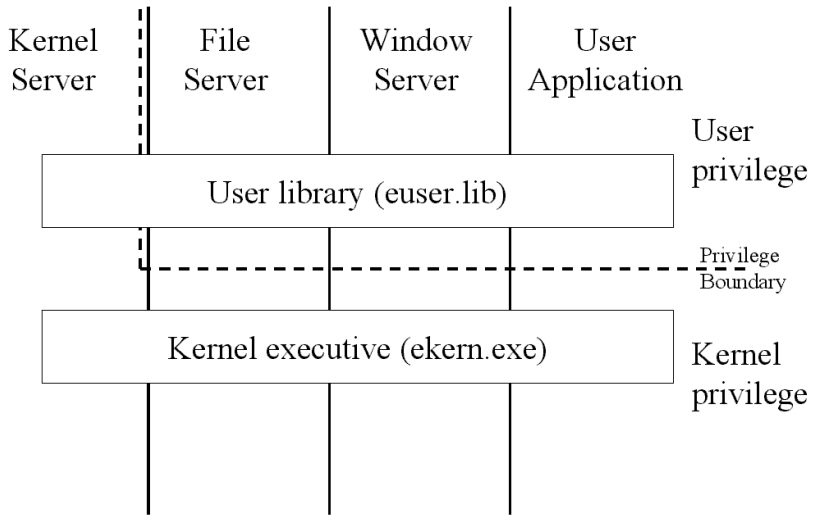

Figure 25. EPOC Kernel and privilege boundaries.

As Figure 25 shows, the kernel executive runs privileged code in the context of the thread that's running. Kernel executive code can therefore be pre-empted by higher priority user-mode threads or the kernel server.

6.2 Application architecture

Considered from CPU side of view, all compiled C++ programs are just series of binary instructions. In order to manage those binaries efficiently they must be packaged. The different packages EPOC supports are:

- Exe programs *(EXE)*.
- Dynamic link libraries *(DLL)*.

Both of these types contain executable program code. The difference between an EXE and a DLL in EPOC is that an EXE is run separately and a DLL is dynamically attached into the program that loads it. DLLs are further divided into separate types. Two most important DLL types are shared library DLLs and polymorphic DLLs.

Shared library DLLs provide a fixed API that has several entry points that the user can call. Programs that use such DLLs are marked so that when they are loaded the system checks if the DLL is already loaded, and if not, the system automatically loads and attaches it. Polymorphic DLLs implement an abstract API such as a device driver or a GUI application. In EPOC, polymorphic DLLs usually have only a single entry point, which allocates and constructs a derived class of some base class associated with that DLL. EIKON applications, for example, are polymorphic DLLs that have an entry point which instantiates the application class that is derived from the *CEikApplication* base class. Polymorphic DLLs usually have a unique postfix in their name to separate them from normal DLLs, for example .app for EIKON applications and .prn for printer drivers.

An executable program has three types of binary data: program data, read-only static data and read/write static data. EXE programs in EPOC are not shared so every time such program is run it gets new areas of memory allocated for all those three types of data. The only exceptions to that are EXEs that reside in ROM. ROM –based EXEs allocate RAM only for read/write program data; the program code and read-only data are read directly from ROM. This is an optimization to save expensive RAM and improve efficiency; ROM –based code is executed in place so no copying is required.

Dynamically loaded link libraries are shared. When a DLL is loaded for the first time, it is reallocated to a particular address. When a second thread requires the same DLL, it is attached to the same copy of the code; no loading is required. A DLL resides in the same memory address in all threads that are using it. EPOC maintains reference counts,

so that the DLL is unloaded when no threads are using it. Because EPOC DLLs are shared, they cannot have writeable static data. This applies to all EPOC DLLs including GUI applications. Writeable static data is not supported because it brings so little benefit in design considering the memory loss and possible error situations it causes /7, p.55/. Static data can be avoided with proper design. In many cases, the design without static data is more robust and modular.

EPOC applications are typically divided into engine, UI, and view components, which are dependent on but logically separate from each other. This improves the modularity and reusability of EPOC applications. The engine contains most of the application logic and it can be shared among applications. The engine is therefore typically packaged as a DLL. This also enables the development of the engine to be easily separated from the user interface and the look and feel design. The EIKON application is a polymorphic DLL, which has only one entry point.

6.3 Active objects

Active objects are an EPOC method to handle asynchronous requests in a modular and effective way. Asynchronous requests mechanism has two main parts: an indicator for the request completion and a semaphore used for informing the thread about the request. Active object design pattern enhances concurrency and simplifies synchronized access to resources /12/.

An asynchronous request status indicates the completion status of a request to a service provider. When an application running in a thread makes a request, it passes an asynchronous request status as a parameter. When the provider completes the request, it stores a success or error code in the request status. Asynchronous request is always between two threads of execution; the requesting thread and the serving thread.

A thread request semaphore is the means by which a provider signals a requester that it has completed a request. The requester can then determine which request has completed, and call the appropriate function to handle the completion of the request.

Static user class methods provide the API to signal a thread request semaphore and to wait until it has been signaled.

An active object encapsulates the general behavior of making requests to asynchronous service providers, and handling the completion of requests. A particular asynchronous service provider typically supplies active object based classes as interfaces by which clients access them.

A thread that uses asynchronous services must have a main loop that waits on the thread's request semaphore for any outstanding requests to complete. The active scheduler encapsulates this wait loop. The main loop of an active object –based program is built around *CActiveScheduler::Start()* and *CActiveScheduler::Stop()* static function calls. All functionality requesting and receiving the actual requests is encapsulated within the active objects and their observers.

An active scheduler is provided by the application architecture for all Graphical User Interface *(GUI)* programs. A high-level view of a GUI application is therefore a set of active objects that handle request completion from events fed to it by its active scheduler. Active objects are used throughout the EPOC. Active objects are used in user interfaces for event handling, communication protocols to handle asynchronous requests, single-thread multitasking and delayed function calls. Active object framework is a good example of the template method –design pattern /1, p 325/. The *CActive* –base class defines a skeleton of the functionality, which is then extended in the derived concrete class. An active object is often used to control some other object structure using the adapter design pattern presented in Chapter 5.5.1.

6.4 EIKON

EIKON applications are typically implemented using the Model-View-Controller architectural pattern presented in Chapter 5.3.2. The MVC pattern enables easy porting of applications between the three different DFRDs. They all have the same structure for UI and event handling; only some DFRD specific features change.

Every EIKON application is a DLL. The application is started by apprun.exe, which loads and attaches it. Then the framework checks that the application has correct UID. Next, the framework calls the first ordinal on the DLL, which must return an object of a class derived from *CApaApplication*. After instantiating the application, the framework calls application class's *CreateDocumentL* method that instantiates the document. Document class is derived from *CEikDocument* base class, which implements some base functionality for document handling. The document class in EIKON application then creates the AppUI, which is derived from *CEikAppUI* class and does the event handling. AppUI then instantiates AppView, that handles the viewing of the document to the user.

The document class in EPOC application is the model in the MVC pattern. The document class presents the data and also has a reference to the engine that in most cases handles the data modification algorithms. AppView is the view in the MVC pattern. The controller handles updates to both the model and the view so in many cases it is programmed as a separate class or into the application class.

The MVC pattern can become particularly blurred in some EPOC applications, because one of the three parts is often missing or two aspects are combined, for example view and controller in smaller applications. However, the MVC architecture is very effective in use and gives so many benefits that it is profitable to use it. The MVC structure enables easy porting of EPOC applications between DRFDs and also swift change of application's Look and Feel *(LAF)*.

6.5 EPOC Idioms

6.5.1 Construction

EPOC has very strict definitions of how to instantiate objects and how to allocate memory. That is due to the fact that the device is practically never turned off and there is a limited amount of memory so one cannot afford allocating much memory and most certainly cant afford loosing any of it with poorly handled memory allocation. C++ has

some weak points in its constructor mechanism if the programmer is not very careful. It is possible to cause memory leaks by allocating memory in class constructor.

EPOC uses two styles of object construction. Normal constructor is allowed only for T and R –type objects that are always used as static objects. Objects that reserve some additional memory for their resources are constructed in two phases. Those classes are always separated from others with the C –prefix and they must be derived from *CBase*. First phase constructor is a normal C++ constructor. Second phase constructor is a class method usually named as *ConstructL()*. Two-phase construction details are usually hidden from the user so that the whole construction is encapsulated inside one static class method called *NewL()*. *NewL* method calls first the normal C++ constructor and then stores the pointer to Cleanup stack. Then it calls the second phase construction and after it succeeds it returns the pointer to the newly created object. If the second phase construction fails the *ConstructL* leaves, the pointer to the class itself is safe in the cleanup stack, and no memory leak occurs since the partly created object can be deleted and memory released.

The construction of a typical C-class looks like this:

```
class CThing : public CBase
      {
      public:           // Construct / destruct
            static CThing* NewL();
            virtual ~CThing();
      protected:        // Construct / destruct
            CThing();
            ConstructL();
      };
```

The user calls *NewL()* to instantiate that class and normal destructor to delete the instances. The real construction methods are protected so that the class can be derived, but the user doesn't have access to them.

6.5.2 Thin templates

Thin templates are an EPOC idiom to avoid code duplication. In thin template pattern, all functionality is provided in a non-typed base class.

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

This base class has the real code so it exists only once. This code is exported from the DLL it resides in. The base class may contain an arbitrary amount of code.

A derived template class is implemented as follows:

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

Because this class uses only inline functions, it generates no extra code. However, because the casting is encapsulated in the inline function, the class is type safe to its users. The derived template class is thin: it generates no new code at all. The user uses the templates as normal template classes. EPOC uses thin templates for example in containers. The details of the idiom are hidden from the application programmer so they can be used like normal C++ STL containers /32/.

Example of EPOC container usage:

```
CArrayPtrSeg<TInt> avararray( 16 );
CArrayPtrSeg<TBool> anotherarray( 32 );
avararray.Insert( TInt( 20 ) );     // works fine
```

```
anotherarray.Insert( TInt( -1 ) );  // does not compile, int can't
                                    // go to boolean array
```

Normal template use would generate separate code for the integer array and Boolean array in the example. With thin template pattern program code exists only once, but still we have type safety for all array types, like integer and Boolean in the example.

7 EPOC APPLICATION DESIGN USING DESIGN PATTERNS

7.1 Introduction

In this chapter, the use of design patterns and EPOC patterns is illustrated with a real world example of application design. The example starts with application description. The work starts with problem domain analysis, which gives the system frameworks and application architecture to start the design. After analysis and requirements we get the architecture and base patterns we are going to use in the application. To that core design, we start adding the features and the patterns and detailed design that implements them.

7.2 Example application

The example application is a non-interactive application called AppTest. AppTest launches another application and tests it by various means. AppTest has no user interface and it is controlled by text scripts. AppTest is a replacement for the system's apprun.exe, and it takes care of all the things that launching an application requires.

7.2.1 Purpose

The purpose of AppTest is to automate feature testing of UI-based applications. Feature testing is also known as black box testing. The term indicates that the application is tested from outside and no modifications are made to the application while it is tested. Black box testing is useful since it tries to do all the things from the same perspective the actual user is using the application. The difficulty in black box testing is that it is impossible to get adequate coverage doing the test manually. Automated testing enables unlimited regression and excellent coverage since the automated test application can permutate through all possible event sequences a UI has. Therefore, an

application to automate black box testing is very important to software development and also improves the quality of testing tremendously.

7.2.2 Features

For a UI-based application, there are several things to test. The basic environmental factors to test are various Out-Of-Memory *(OOM)* and Out-Of-Resources *(OOR)* situations. The testing application has to be capable of sending UI events to the tested application to simulate the user using the application. AppTest also tries to monitor the state of the tested application and observe if the tested application hangs or exits abnormally. AppTest writes a log of test procedure and of the tested application's behavior.

7.3 Requirements

Requirements for AppTest are easily derived from its features. A certain set of requirements also comes from the EPOC environment and its properties. The software development process also sets some boundaries and requirements. Some of the EPOC requirements and boundaries are presented in Chapters 4.3 and 6.

AppTest is used to test another application so obviously it must be very stable and affect the operating system and resources as little as possible. If AppTest itself would be very resource consuming it would most certainly affect the tested application and would either lead to abnormal behavior of the tested application or misinterpreted test results.

The used software process will be incremental and iterative. Each increment adds some functionality to a working application from the previous increment. The design must be as flexible as possible so that features can be added, modified and removed at any phase of the development.

At some point, a user interface might be added to improve usability. This forces the design to be modular and not strictly tied to the environment. The design should also provide reusable classes and components for future use and development. At the end of the requirement analysis, we have a set of features we want to include in the application. We also have to check that the features are not in contradiction against each other.

7.4 Analysis

The requirements set quite strict limits for the analysis and design of the application. On the other hand, EPOC has also limitations and restrictions that we have to deal with. Before starting to analyze the problem domain in class or pattern level, we have to make some architectural decisions. The architecture we choose to use is very often dictated by the environment and the size and type of the application we are building.

The analysis on the application domain is an iterative process illustrated in Figure 26. First, we prioritize and categorize the requirements. Then we take a requirement, check out the environment restrictions, design restrictions and if it passes the restriction analysis then we try to find a pattern that best encapsulates that behavior. After that, we add that pattern to our skeleton design and check it's effect on the whole design. A pattern may solve several requirements at once or make some requirements impossible to fit in the design. A pattern also adds, removes and modifies the restrictions for further features.

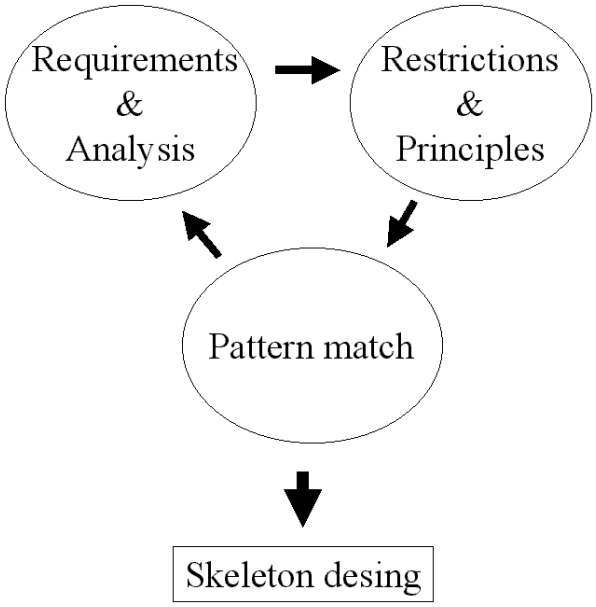

Figure 26. Pattern finding cycle.

AppTest does not have a UI nor does it use any other external modules besides the EPOC kernel. All parameters are read from configuration scripts. This enables us to encapsulate the whole functionality inside one executable.

AppTest has to monitor events that happen asynchronously on the other application. EPOC has a few different ways to deal with asynchronous requests and services. The possibilities are to use wait-loop programming, callback functions, or active objects. The most effective and modular solution is to use active objects and active scheduler. The use of active object framework forces our application skeleton into a certain form and also sets requirements for all other features we are to implement into our application.

We have a basic requirement that our application has to be aware about the state of the other application that is under test. AppTest has to monitor certain flags and incidents to be aware about the state of the application. The incidents that cause the state changes vary in different states so the behavior of AppTest must also vary. *RThread* is the main class that is used to analyze, monitor and control the application.

AppTest will monitor another application that is executed in a separate thread. EPOC has an API for monitoring the state of a thread. This API has a class *RUndertaker* that we are going to use to inform us that the tested application thread has exited. During the automated test, the tested application may end up in a deadlock for some reason. Therefore we have to instantiate a timer that stops the test and kills the application if it has hung. *RTimer* is a basic EPOC timer class that can be used for this.

To do the actual black box testing we need to send UI events to the tested application. The sending is done according to a schedule that is recorded from the actual use of the tested application. To play the recorded sequence, we need a timer and some framework dependent classes to send the events of different type via proper interfaces.

The decision to use Active scheduler and active objects as a basis for the application forces us to encapsulate those R-classes inside active objects so we will have to define corresponding C-classes derived from *CActive*.

7.4.1 Selecting patterns

The functionality of AppTest changes as a function of its state. This kind of behavior is best implemented by using the state design pattern. State design pattern's intention is to allow the change of behavior when internal state changes. We have several phases in the application that we are trying to monitor and several different actions to take in each phase. In the beginning of the development we are also not sure that how many phases the tested application has in its lifecycle because we are not sure how deep in detail we can monitor the tested application through EPOC kernel APIs. State pattern as well enables easy addition of new states without breaking down the existing code. State also fulfills the OCP design principle discussed in Chapter 4.2.1. We program the states we know for sure, which are initialization, running and terminating. If we later find out that we can monitor the application more in detail, we can add new states without having to change the behavior of existing states.

The actions in different phases are done by *CActive* -based active objects described in the previous chapter. To implement asynchronous services provided by R –classes we need to adapt them to the active object framework. Adapter design pattern transforms existing functionality into a new interface or environment. Adapter enables us to wrap the existing R –class functionality into more robust and intelligent active object framework without having to write enormous amounts of new code.

The application has several active objects and several objects that are interested in their state changes externally. Observer design pattern enables objects to inform other objects about their state changes. Observer pattern also allows us to vary the observers of certain subjects in run time, which is quite crucial when we have different subjects active in different states of the program run.

7.5 Design

From the analysis phase, we have the main design patterns and the skeleton of our design. The application will have its main loop around the active scheduler and the state design pattern. The main actors in the application will be the active objects. The states form an asynchronous state machine, which changes state as a function of active objects. The active objects are the subjects for states and the states work as observers to active object's activity.

7.5.1 Adapter

We need to adapt R –classes into the active object framework. This is done using the adapter pattern presented in Chapter 5.5.1. We use object adapters since R –classes are meant to be used as references, not derived. *CActiveScheduler* is the user, *CActive* is the target class, *CdUndertaker* is the adapter and *RUndertaker* is the adoptee. *CdUndertaker* has similar structure. Figure 27 shows the *CdUndertaker* implementation of the adapter design pattern.

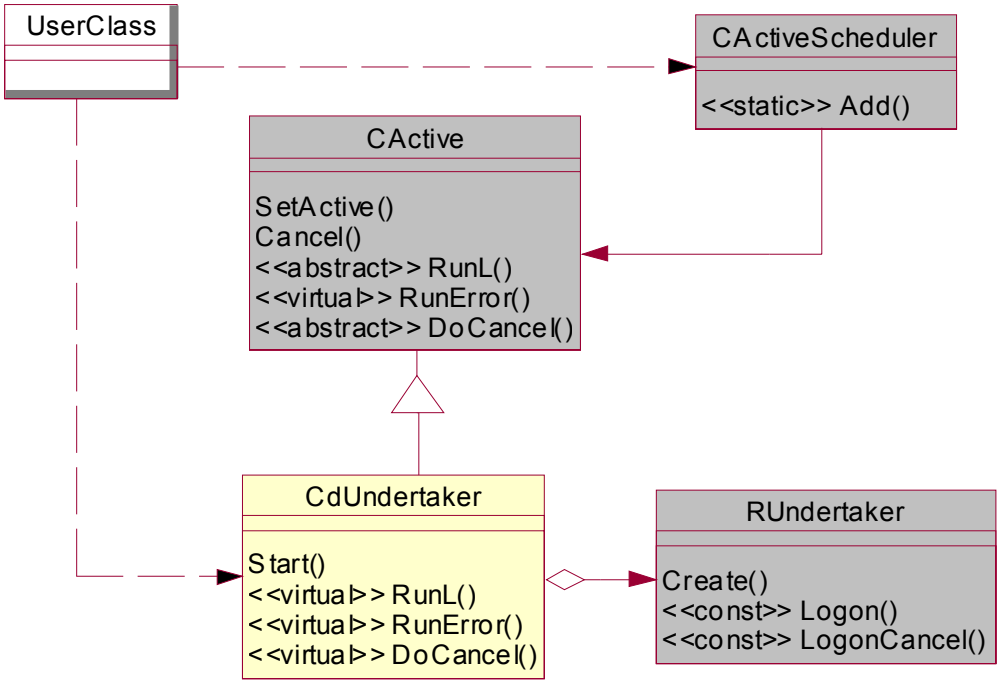

Figure 27. Adapter Design Pattern in AppTest.

7.5.2 State

State is used to improve flexibility of our design and to enable later additions or changes to the monitoring logic. The state machine is implemented into *CdAppTester* class. *TState* is the abstract base that defines the interface and contains the reference to the state machine represented by the *CdAppTester*. Concrete state classes TStateInit, TStateRun and TStateTerminate only add their own implementations for the *EnterL()*, *Continue()* and *Cancel()* methods. *EnterL* is called when the state is entered and it initializes the asynchronous requests the state uses. *Continue* is called when some of the asynchronous request that the active states *EnterL* made is served. *Continue* returns the reference to the next active state so that the state machine proceeds. Active states *Cancel* gets called if the user wants to stop the state machine. In the beginning the *TStateInit* is the active state so its *EnterL* is called. When one of the active requests it makes is served, it returns the next active state in *Continue*. AppTest implementation of state pattern is shown in Figure 28.

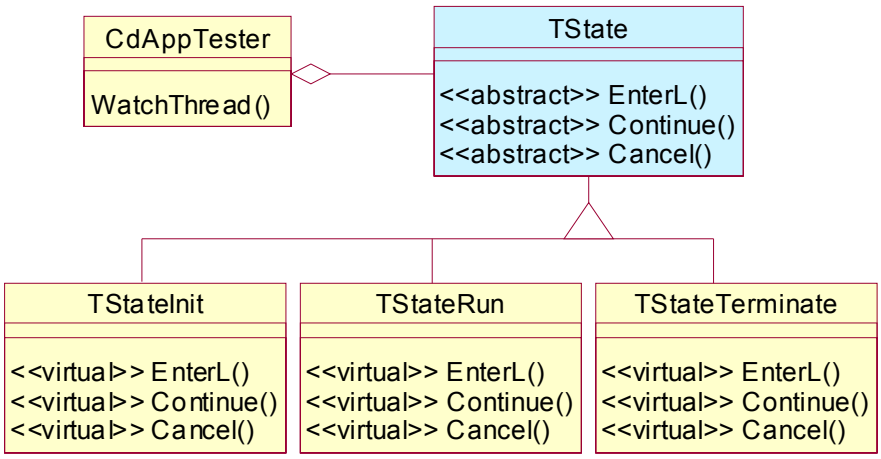

Figure 28. State Design Pattern in AppTest.

7.5.3 Observer

Observer pattern is used to map the many-to-many relationship between different states and different active objects. Each state may have several active objects making requests and when any of the requests is fulfilled, the calling state must be informed. Observer pattern makes it fairly easy to implement this behavior. The state classes are derived from the observer interface and the active objects from the subject interface. The subjects keep track on the active observers registered to them and notify them when their inner state changes. When *CdGrimreaper* for example activates, it informs the *TStateRun* and the *TStateRun* can move the state machine into next active state, which in this case would be *TStateTerminate*. *TdMessage* is used to retain the information about the subject that was activated. This is done with a flag object because EPOC does not have RTTI and does not therefore support dynamic determination of the actual subclass as discussed in Chapter 5.6.1. The use of observer in AppTest is shown in Figure 29.

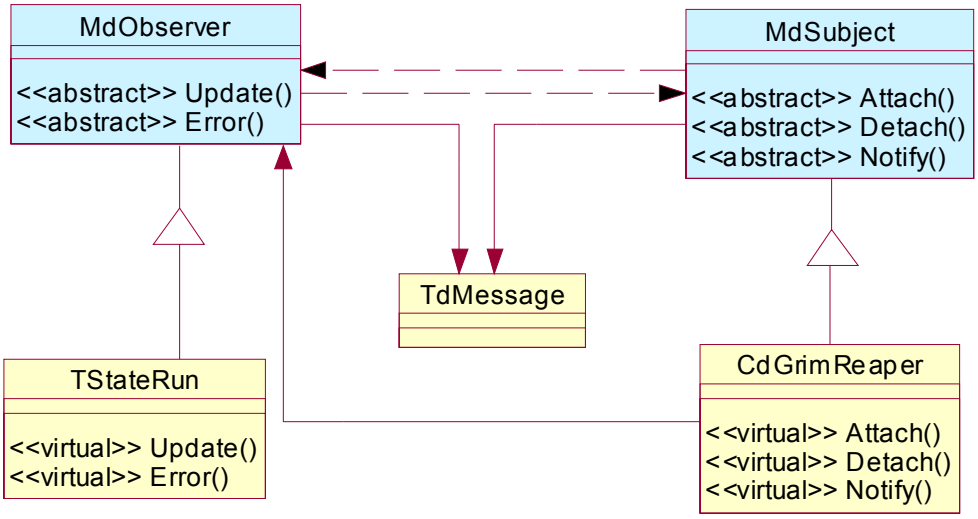

Figure 29. Observer Design Pattern in AppTest.

7.5.4 Factory Method

To send the events we need a *RTimer*-based object that can play different types of events. The easiest way is to define an abstract base class for events and to implement an active object that adapts RTimer into the active scheduler. We also need to separate the creation of the different concrete event types from the playback. Factory method is a good choice when it is needed to encapsulate the instantiation of the concrete types. Factory method also allows us to add new event types later without changing the *CdEventPlayer* class or any its dependent. This clearly adds expandability and makes the design follow the OCP design principle. Figure 30 shows the factory method used in AppTest.

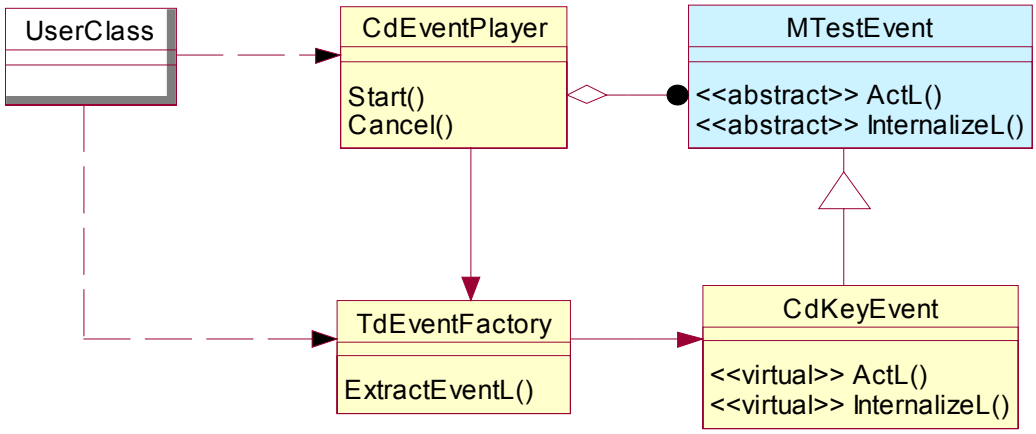

Figure 30. Factory Method Design Pattern in AppTest.

7.5.5 Singleton

The application has several services that are needed globally throughout the application. Since the application is first built as an EXE without a UI, it is possible to use Singleton pattern to enable global access to those resources. The singleton can be implemented either by using normal form /1//16/ or by a more convenient template based solution /28/. The template version of singleton can be seen in the top right corner of the final design of the AppTest in Figure 31.With singleton, we could remove most of the illogical references between collaborating classes thus making them more separate and easier to reuse.

When the application will have a user interface, it will most certainly be changed into an APP so it can no more have writeable static data as explained in Chapter 6.2. That restriction will disable the standard singleton pattern implementation and either forces us to implement the singleton using a global registry or to remove the singleton from the design. Using a global registry to reference another object is a slow operation and can therefore be forgotten. Removing the singleton would get us back to the situation where we were before we added the singleton so it is better to forget the singleton in the first place.

7.6 Final Design

Figure 31 presents the final design of the AppTest combining all the patterns mentioned earlier and a few other basic structures. The final design is a pattern system having several interesting parts. State, observer and adapters are combined to work as an asynchronous event based system. Factory method adds extensibility to the design. The design is very modular, entities working in different tasks can be clearly seen and thus easily traced. Different patterns are easy to see from the diagram.

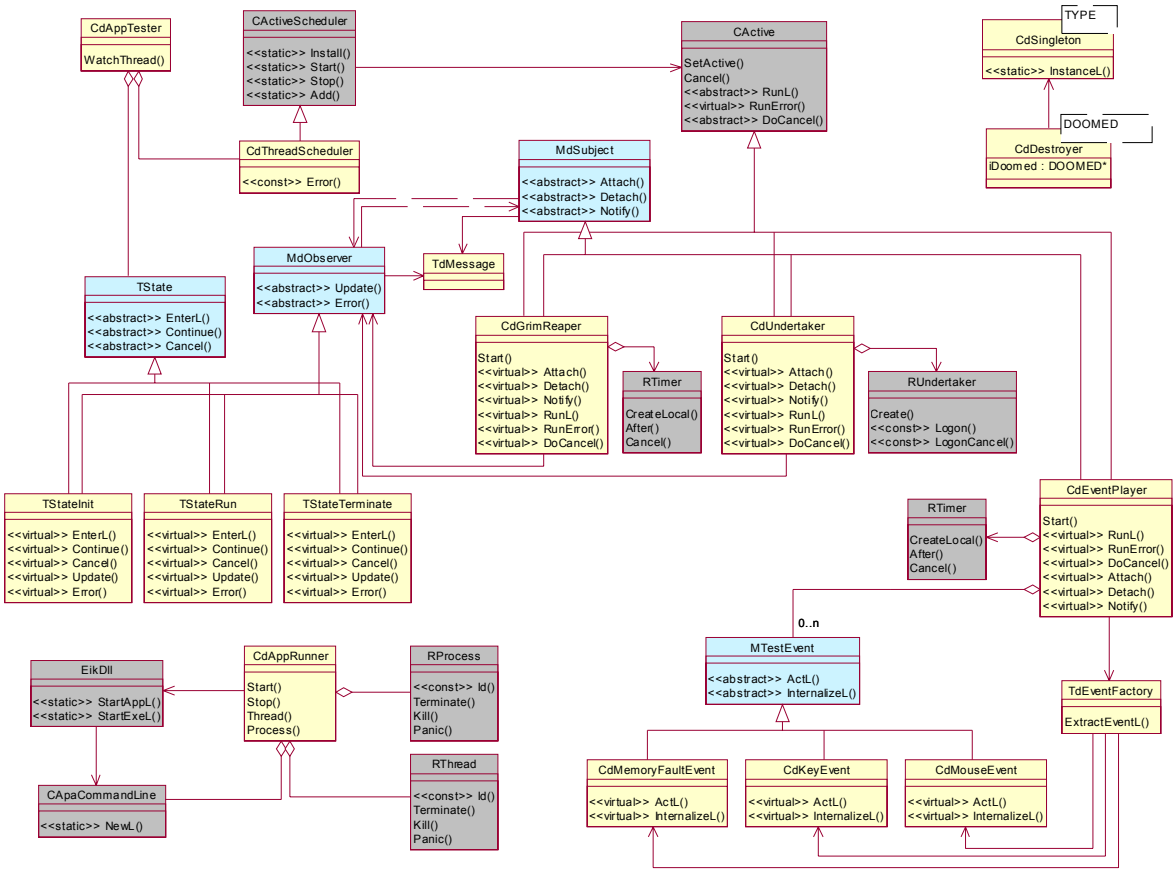

Figure 31. AppTest class diagram.

7.7 Discussion

The design was built up rapidly with the aid of design patterns. The abstract interface classes make it very easy to enhance existing features and add new ones by deriving new classes. Patterns lead into well-encapsulated design where it is safe to modify one part and be sure that the rest will keep working.

The collaborations between classes are specified using those abstract interfaces so the testing of functionalities is uniform. When one test case is ready, it can be reused to define the rest with little modifications. Only one unit test structure is needed for every collaboration, all derived subclasses use the same form of test.

Analyzing the design, we see that OCP and DIP are followed very well. The design is easily expandable at many places thanks to the abstract interfaces. Single class entities can quite easily be reused since the ISP is also fulfilled in good degree. Most entities work in several roles, defined by separate interfaces. ADP is somewhat followed since concrete classes depend on abstract base classes, not from other concrete classes. Classes handling the asynchronous requests have the subject interface to their observers and adapted *CActive* interface to the framework. They can therefore be reused either as subjects in another context, or active objects. *TState* based objects have similarly a double role and thus greater reusability.

The use of design patterns in this application compressed time from both analysis and design phase and improved the design making it more reusable, flexible and expandable. Design patterns also made is simpler to test the classes and structures.

8 CONCLUSIONS

The adaptation of design patterns is fairly easy for any software designer who is familiar with UML or similar modeling language used to describe the patterns. Many good designers who have not ever heard of design patterns often find out that they have been using them unconsciously. The following conclusions are based on the experiences gained during the work of this thesis.

Design patterns provide an excellent method to speed up and simplify the development process. However, the statement “owning a hammer does not make one an architect” applies to design patterns as to all other object-oriented design methods and tools /4/. To properly use design patterns and to get the benefits from their usage requires studying and experimenting with them in different problem domains. A perfect pattern for one problem may prove to be a total disaster in another design even in a similar problem domain. In the implementation phase of this thesis the adapter, state and observer patterns proved to be very usefull in an EPOC application.

A design or a design pattern is only a model that describes the structure and collaborations in an abstraction level. The design must be implemented using some implementation methods and even the greatest design will fail if the implementation is not done correctly and the design is not understood clearly enough. The implementors have to understand the patterns to know the implementation tricks and to avoid ruining the pattern by improper implementation.

The environment sets the strictest principles for the design to fulfill. Those principles must be met in design and implementation. This fact is also a good starting point in analyzing a design pattern and making conclusions of it usability in a problem domain. If the pattern fails to fulfill for example EPOC restriction on writeable static data or cleanup stack usage the pattern is not suitable for EPOC environment even if it would be perfect in some other environment as was seen with the singleton pattern in this thesis.

General design principles are also good candidates when analyzing a software design. The design principles have evolved through years or decades of software programming and therefore have a solid base in defining what works and what often fails. The more design principles the pattern can meet the better it will eventually prove to be. Using those principles and other common software process metrics, it is quite easy to prove that design patterns, when used by experienced software designers, really speed up and simplify the design process and also lead to better quality.

An object-oriented environment, such as EPOC using patterns in frameworks and APIs forces the developers to familiarize themselves with those patterns. A misuse of system structures will eventually lead into nonworking application, bad user experience and possible crashes. On the other hand, after identifying the patterns and using them frequently the collaborations and structures inside them start reflecting into developers own application designs leading into better quality, response times and usability. Active object framework and the MVC pattern were discovered very useful during the work of this thesis.

However, design patterns do not make a poor design excellent nor do they make a good design fail. There are no “cookbook” methods that can replace intelligence, experience and good taste in design and programming /13/. As the inventor of the C++ language, Bjarne Straustrupp wisely said: “Design and programming are human activities; forget that and all is lost”.

9 REFERENCES

/1/ Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object Oriented software. Addison Wesley Longman, Inc 1995. 19th printing, January 2000. ISBN 0-201-63361-2.

/2/ Frank Buschman, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. Pattern oriented software architecture: A System of Patterns. John Wiley & Sons Ltd 1996. 6th printing June 2000. ISBN 0-471-95869-7.

/3/ Kimmo Hoikka. Design Patterns in EPOC Software Development. Presentation at Symbian Developer Expo 6th November 2000, London. Available: http://www.symbiandevnet.com/techlib/techcomms/transcripts/uk2000/technology.html .

/4/ Craig Larman. Applying UML and Design Patterns: An Introduction to Object-Oriented Analysis and Design. Prentice Hall, Inc 1998. ISBN 0-13-748880-7.

/5/ OMG Unified Modeling Language Specification. Version 1.3, June 1999. Available: http://www.omg.com/uml/ [Referenced 20.1.2001].

/6/ Digia Coding Conventions. [Confidential]. Available: Digia Intranet.

/7/ Martin Tasker. Professional Symbian Programming: Mobile Solutions on the EPOC platform. Wrox Press Ltd. First printing, 2000. ISBN 1-861003-03-X.

/8/ Symbian website [Internet]. Location: http://www.symbian.com/ [Referenced 25.2.2001].

/9/ Psion, Inc. [Internet]. Location: http://www.psion.com/. [Referenced 26.2.2001].

/10/ Nokia, Oyj [Internet]. Location: http://www.nokia.com/. [Referenced 25.2.2001].

/11/ Digital Information Architects, Digia Inc [Internet]. Location: http://www.digia.com/. [Referenced 25.2.2001].

/12/ Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschman. Pattern-oriented Software Architecture volume 2: Patterns for Concurrent and Networked Objects. John Wiley & Sons Ltd 2000. ISBN0-471-60695-2.

/13/ Bjarne Straustrupp. The C++ Programming Language, Third Edition. Addison Wesley Longman, Inc 1997. 8th printing, October 1998. ISBN 0-201-88954-4.

/14/ James O. Coplien. Multi Paradigm Design in C++. Addison Wesley Longman, Inc 1998. ISBN: 0-201-82467-1.

/15/ Kayshav Dattatri. C++: Effective Object-Oriented Software Construction: Concepts, Principles, Industrial Strategies, and Practices, Second Edition. Prentice Hall, Inc 2000. ISBN 0-13-086769-1.

/16/ Bruno R. Preiss. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. John Wiley & Sons Ltd 1999. ISBN 0-471-24134-2.

/17/ Design Principles and Design Patterns. Robert C. Martin. Available: http://www.objectmentor.com/publications/articlesBySubject.html [Checked 26.2.2001]

/18/ Robert C. Martin. Open Closed Principle. C++ Report January 1996. Available: http://www.objectmentor.com/publications/articlesBySubject.html [Checked 26.2.2001].

/19/ Robert C. Martin. Dependency Inversion Principle. C++ Report May 1996. Available: http://www.objectmentor.com/publications/articlesBySubject.html [Checked 26.2.2001].

/20/ Robert C. Martin. Interface Segregation Principle. C++ Report August 1996. Available: http://www.objectmentor.com/publications/articlesBySubject.html [Checked 26.2.2001].

/21/ Robert C. Martin. Acyclic Dependency Principle. C++ Report November 1996. Available: http://www.objectmentor.com/publications/articlesBySubject.html [Checked 26.2.2001].

/22/ Robert C. Martin. Design Patterns for Dealing with Dual Inheritance Hierarchies. C++ Report April 1997. Available: http://www.objectmentor.com/publications/articlesBySubject.html [Checked 26.2.2001].

/23/ Robert C. Martin. Acyclic Visitor: A design pattern for eliminating dependency cycles in Visitors. Available: http://www.objectmentor.com/publications/articlesBySubject.html [Checked 26.2.2001].

/24/ Martin Fowler. Refactoring: Improving the Design of Existing Code. Addison Wesley Longman, Inc 1999. 4th printing, February 2000. ISBN 0-201-48567-2.

/25/ Symbian. Quartz Version 6.0 Edition for C++. [CD-ROM].

/26/ Symbian Devnet [Internet]. Location: http://www.symbiandevnet.com/. [Referenced 25.2.2001].

/27/ The reason why AriadneV exploded. [Internet] Location: http://users.deltanet.com/~tegan/ariane.html. [Referenced 30.11.2000]. Also available at: http://www.student.math.uwaterloo.ca/~cs445/handouts/lectureSlides/W01/documents/se_re.pdf [Checked 26.2.2001]

/28/ John Vlissides. Pattern Hatchling: Design Patterns Applied. Addison Wesley Longman, Inc 1998. ISBN 0-201-43293-5.

/29/ Christine Hofmeister, Robert Nord, Dilip Soni. Applied Software Architecture. Addison Wesley Longman, Inc 2000. ISBN 0-201-32571-3.

/30/ Scott W. Ambler. Process Patterns: Building Large-Scale Systems Using Object Technology. Cambridge University Press 1998. ISBN: 0-521-64568-9.

/31/ Martin Fowler. Analysis Patterns: Reusable Object Models. Addison Wesley Longman, Inc 1997. 6th printing, December 1998. ISBN 0-201-89542-0.

/32/ Matthew H. Austern. Generic Programming and the STL: using and extending the C++ Standard Template Library. Addison Wesley Longman, Inc 1998. ISBN 0-201-30956-4.

/33/ Yun-Tung Lau. The Art of Objects: Object-Oriented design and Architecture. Addison Wesley Longman, Inc 2001. ISBN 0-201-71161-3.