

# 1

## Introduction

### 1.1 About this Book

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

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

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

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

### 1.2 Who this Book Is For

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

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

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

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

### 1.3 Which Version of Symbian OS this Book Is For

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

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

### 1.4 General Design Patterns

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

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

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

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

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

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

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

## 1.5 Symbian OS Patterns

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

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

### 1.5.1 Constraints on Software Based on Symbian OS

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

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

---

<sup>1</sup>[www.symbian.com/news/pr/2008/pr20089781.html](http://www.symbian.com/news/pr/2008/pr20089781.html).

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

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

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

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

## 1.5.2 Important Forces in the Context of Symbian OS

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

- *Minimal use of RAM*  
Random Access Memory (RAM) is a type of computer data storage. A key characteristic of RAM is that it is volatile and information it contains is lost when a device is turned off. The three main contributors to RAM are:
  - Call stack – a dynamic data structure which stores information about the current execution including details of the active functions and local data
  - Heap – an area of memory that you can dynamically allocate from using the `new` operator; ‘free store’ is another name for a heap
  - Code – devices that are not eXecute-In-Place (XIP) based load code from secondary storage into RAM where it is executed. Exactly how this is done depends on the memory model used but code can make up a significant proportion of the RAM used on a device.

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

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

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

- *Minimal use of secondary storage*

Secondary storage is where code, read-only data and persistent data are stored. Most Symbian OS devices support secondary storage by providing an area of flash memory<sup>2</sup> though there exist some that have a hard disk drive.

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

- *Minimal execution time*

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

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

- *Real-time responsiveness*

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

---

<sup>2</sup>[en.wikipedia.org/wiki/Flash\\_memory](http://en.wikipedia.org/wiki/Flash_memory).

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

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

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

- *Reliability*

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

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

- *Security*

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

### 1.5.3 Other Forces

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

- *Development effort*

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

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

- *Testing cost*  
Software testing is the process used to assess the quality of your software. It is important to remember that, for most practical applications, testing does not completely establish the correctness of a component;<sup>4</sup> rather, it gives you a level of confidence in how reliable your software will be. With this in mind, it is important to design your software so that testing is made as easy as possible to allow you to quickly and easily establish the assurance you need to ship your software.
- *Maintainability*  
This is defined by [ISO 9126] to be ‘the ease with which a software product can be modified in order to correct defects, meet new requirements, make future maintenance easier or cope with a changed environment’. Before you invest too heavily in this, hence increasing your development effort, you need to be sure that your product will benefit from it. For instance, if you only wish to target UIQ devices then making your application UI highly portable isn’t necessary. Of course that assumes you don’t change your mind later!
- *Encapsulation*  
This is the ability to provide the minimum necessary well-defined interface to a software object in such a way that the internal workings are hidden. A well-encapsulated object will appear to be a ‘black box’ for other objects that interact with it without needing to know how it works. The goal is to build self-contained objects that can be plugged into and out of other objects cleanly and without side effects.  
Improving the encapsulation of your objects helps with reliability because objects do what you expect them to do. It also helps with maintainability because you minimize the risk that extending or changing the software breaks other parts of the application or service. It also reduces testing cost by allowing you to more easily isolate an object for unit testing. However, this is at the expense of reduced flexibility.

#### 1.5.4 Pattern Elements that Are Already in Place

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

---

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

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

## 1.6 Design Pattern Template

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

### Pattern Name

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

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

### Problem

#### *Context*

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

#### *Summary*

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

#### *Description*

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

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

### ***Example***

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

## **Solution**

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

### ***Structure***

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

### ***Dynamics***

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

### ***Implementation***

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

### ***Consequences***

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

### ***Example Resolved***

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

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

## Other Known Uses

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

## Variants and Extensions

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

## References

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

## 1.7 Structure of this Book

The patterns in this book cover the following topics:

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

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

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

## 1.8 Conventions

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

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

```
This is example code;
```

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

## 1.9 Other Sources of Information

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

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

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

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