

# Long Running Active Object: a Design Pattern

Ian McDowall

Published by the Symbian Developer Network

Version: 1.0 – September 2008

|          |                                   |           |
|----------|-----------------------------------|-----------|
| <b>1</b> | <b>INTRODUCTION</b> .....         | <b>2</b>  |
| <b>2</b> | <b>THE DESIGN PATTERN</b> .....   | <b>2</b>  |
|          | 2.1 INTENT .....                  | 2         |
|          | 2.2 THE PROBLEM .....             | 2         |
|          | 2.3 THE SOLUTION .....            | 4         |
|          | 2.4 OTHER KNOWN USES .....        | 12        |
|          | 2.5 VARIANTS AND EXTENSIONS ..... | 12        |
|          | 2.6 REFERENCES .....              | 12        |
| <b>3</b> | <b>CONCLUSION</b> .....           | <b>12</b> |
| <b>4</b> | <b>AUTHOR PROFILE</b> .....       | <b>13</b> |

# 1 Introduction

This article describes a commonly used design that solves the problem of allowing a thread to perform a long running background task whilst still remaining responsive to higher priority events. The article is laid out as a design pattern, in a style similar to the forthcoming Symbian Press book, *Common Design Patterns for Symbian OS*. More information about the book can be found at [developer.symbian.com/designpatternsbook](http://developer.symbian.com/designpatternsbook).

The aim of all design patterns is to capture the collective experience of skilled software engineers based on proven examples to promote good design practices. Design patterns are 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 in hand. Only rarely will a problem be solved in an entirely unique fashion. The first chapter of the *Common Design Patterns for Symbian OS* book goes into this subject in more detail and is also available as a PDF from the official website at [developer.symbian.com/designpatternsbook](http://developer.symbian.com/designpatternsbook).

This article is suitable for relative beginners as well as experts in Symbian OS development. As with any design pattern, 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 design pattern described here should help both groups to harness a well-proven solution, as well as any specific variations, to solve individual design problems.

To get the most out of this article, readers are expected to have some existing knowledge of Symbian OS and Symbian 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 (for example, C, R, T and M classes) and how they behave.

If you are new to Symbian OS or want to take a refresher course in these concepts, then you'll find suitable material at [developer.symbian.com/main/learning/fundamentals/index.jsp](http://developer.symbian.com/main/learning/fundamentals/index.jsp). We also recommend that you consult the Symbian Developer Library documentation at [developer.symbian.com/main/documentation/sdl](http://developer.symbian.com/main/documentation/sdl) and view the range of Symbian Press publications listed at [developer.symbian.com/books](http://developer.symbian.com/books).

## 2 The Design Pattern

### 2.1 Intent

Ensure a thread remains responsive during a long running task by using an active object to split it up into manageable chunks that are prioritized against any incoming events.

### 2.2 The Problem

#### 2.2.1 Context

A process needs to carry out a long-running task while remaining responsive to high priority events.

### 2.2.2 Summary

- The task to be carried out is long-running or has an unpredictable duration.
- The task to be carried out may need to be cancelled before completion.
- The developer does not want to create a separate thread for the task, either to minimize resource usage or to avoid the complexity of inter-thread communication.

### 2.2.3 Description

The problem this pattern is intended to solve is that of a process which has to carry out a number of tasks that may take a significant amount of elapsed time. Some of these tasks may be obviously long-running (such as sorting or compacting large amounts of data or fetching and rendering a Web page) but others may only become long-running under certain circumstances (such as resources that need to be fetched over an external connection or from a low-speed device).

Symbian OS provides preemptive multi-processing between threads but a single long-running task can still cause problems if it is not designed properly. For instance, a long-running task which is running within a single thread that is also visible to the end user requires some form of progress or activity indication to prevent the end user from becoming frustrated. If a task takes longer than the end user expects he or she may wish to cancel the task, which then needs to be handled promptly to properly close and free up any resources being used in the task.

One alternative to this is to use two threads within a single process: one to handle high priority events whilst the other continues the long-running task in the background. By assigning appropriate priorities to these two threads, it is possible to remain responsive while still completing the background task. Symbian OS supports multithreading such as this and provides mutexes, semaphores, critical sections, etc. to enable you to control the interactions between threads.

However, whilst this does allow the process to show progress notifications, co-ordinate the threads and handle events from the end user, this multithreaded design is difficult to implement robustly. In addition, by introducing additional threads into the process, you increase the resource usage of your solution considerably as a thread by default uses at least 17KB of RAM.

### 2.2.4 Example

Any task that is long-running or whose time to run is unpredictable is a candidate for this pattern. For illustrative purposes, we will consider a process that takes a file containing an indeterminate number of vCard contact entities that need to be imported into the Contacts database. We will put to one side for now the question of how the vCard file is obtained (which might itself be another long-running task).

The following code fragment is given a file name and an open handle to the Contacts database. It opens the file, reads the contents into a buffer and then imports all of the contacts in one operation.

```
TBool ImportAll(CContactDatabase* aCdb, const TDesC& aFilename)
{
    // Make a read stream based on a buffer with data read from file
    RFs rfs;
    User::LeaveIfError(rfs.Connect());
    CleanupClosePushL(rfs);
    RFile inFile;
    err = inFile.Open(rfs, aFilename, EFileRead | EFileShareReadersOnly);
    User::LeaveIfError(err);
    CleanupClosePushL(inFile);

    CBufBase* buffer = CBufFlat::NewL(KReadSize);
```

```

CleanupStack::PushL(buffer);
RBufReadStream readStream;
TBuf8<KReadSize> fileDes;

do
{
    err = inFile.Read(fileDes, fileDes.MaxLength());
    if((err == KErrNone) && (fileDes.Length() > 0))
    {
        buffer->InsertL(buffer->Size(), fileDes);
    }
} while((err == KErrNone) && (fileDes.Length() > 0));
CleanupStack::PopAndDestroy(2); // inFile, rfs

readStream.Open(*buffer);
CleanupClosePushL(readStream);

// Import all the contacts using the data read from the file.
TBool importOk = EFalse;
CArrayPtr<CContactItem>* entryArray = aCdb->ImportContactsL(
    TUid::Uid(KUi dVCardConvDefaultI mpl),
    readStream, importOk, 0);

delete entryArray; // We don't check what has been imported

CleanupStack::PopAndDestroy(2); // readStream, buffer

return importOk;
}

```

All of the operations used here are synchronous but they are not necessarily short or predictable:

- If the file provided contains a large number of entries then clearly the import operation will take a long time.
- Even if the file contains only a small number of entries, importing them into the database will occasionally take a significant period of time if the database needs to be compacted as a result of the insertion.

As this code would be executed with the thread of a user-side application, it would cause that thread to become unresponsive for an indefinite period of time. This is not the fault of the Contacts model but rather is a fault in the way it is being used.

## 2.3 The Solution

Split up the task into a sequence of stages or states that can be completed sequentially and then use `CIdle` or another `CActive`-derived base class to execute them within a single thread. By using the Active Object pattern we make use of the Symbian OS active object framework to enable efficient multitasking within a single thread, hence allowing you to support progress reporting, cancellation and pre-emption more easily and without resorting to multithreading.

### 2.3.1 Structure

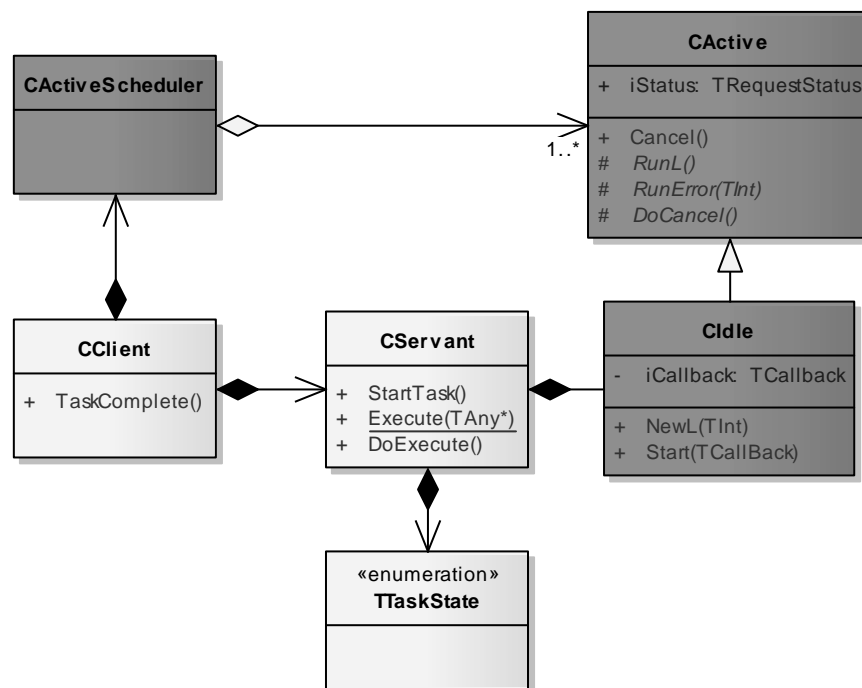
The long-running task is broken up into smaller stages and an active object is created to be responsible for it. The active object performs the task by carrying out each stage in turn before

queuing itself to carry out the next stage. At each stage, the active object can update progress information as well as carrying out the main task. Using an active object also allows an external process to easily cancel the active object if necessary.

For tasks that involve retrieving or processing a large amount of data, this structure will be quite natural because the APIs used cannot afford to handle all of the data in one operation. However, for other types of task, it may require more design effort to split the overall task into a sequence of stages. Either way, there are three classes employed in this structure:

- A single class, known as the *servant*, is used to implement the overall task; the servant's responsibilities are divided up into a series of states that represent each of the stages defined in the overall task.
- The servant uses the *CIdle* class provided by Symbian OS to handle all the machinery to continually reschedule the active object.
- The final class in this structure is the *client* class that creates the servant and initiates the task.

This structure is shown in Figure 1:



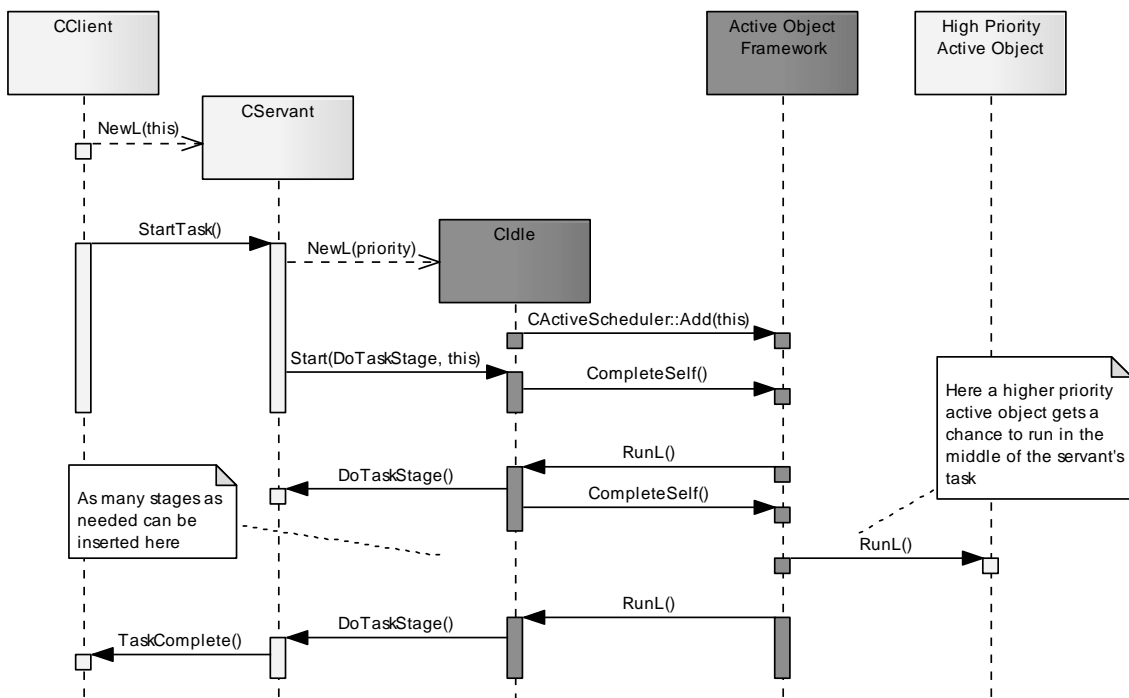
**Figure 1: Long Running Active Object Pattern Structure**

### 2.3.2 Dynamics

The key dynamic aspect of this pattern is that the task under analysis is broken up into stages which are each executed within the context of the *CIdle*'s `RunL()`. However, between each call to `RunL()` there is the opportunity for other, higher priority active objects to be called to respond to more important events.

In Figure 2, the `CompleteSelf()` message represents the steps taken by the *CIdle* object to register itself with the active object framework as being ready to run immediately. Whether or not it

is run immediately then depends on what other active objects are in the active scheduler's queue that are also ready to run.



**Figure 2: Long Running Active Object Pattern Dynamics**

### 2.3.3 Implementation

The implementation of this pattern has the following key features:

- The servant must provide a way for the client to prime it with all the information required to execute the long-running task.
- A TCalIBack object is used by the servant to encapsulate the function that performs the long-running task. This function must be a static function and takes just a single TAny\* parameter. The TCalIBack object is given to the CIdle object to call each time it runs.
- Since this function is called a number of times to achieve a single object it must maintain some internal state. This is achieved by providing a static function on the servant class, such as Execute(TAny\*), that takes a pointer to the servant as the argument. The servant must then track the progress of the task so that each stage of the task builds on the previous one. Progress displays or cancellation checks should be built into the execution of the task where applicable.
- The servant will normally require a means of signaling the completion of the whole task back to the client. In some cases, this isn't necessary if the client wishes to treat the task in a 'fire and forget' manner.
- The priority of the task is reflected in the priority given to the CIdle object when it is created by the servant.

The main thing to consider during implementation is how to split up a long-running task. In general, there are two main ways of achieving this:

- Use a sequence of states.

With this approach, a sequence of states is created and the servant simply moves between them. The sequence of states must be fixed in advance and finite in number. This approach is appropriate when a task requires a number of different operations to be carried out.

- Simply iterating to completion.

This approach is appropriate when the task involves repeating a similar step many times, especially if the number of steps that will be required isn't known at the time you implement this pattern.

When considering how much work should be carried out in each stage, you need to balance responsiveness with efficiency. If a stage takes longer than about 100ms, then the end user may start noticing it and the UI will be disrupted. However, if the stage is too short, then the overheads of the implementation can become significant (even though using `ClIdle`, and active objects in general, are quite efficient). This consideration is particularly relevant when a task is split up in the second way, that is, by iterations.

This balance can be very difficult to achieve in advance by analysis and it is probably wise to make the amount of work in each stage configurable in some way so that it can be tuned during development or even later.

## 2.3.4 Consequences

### 2.3.4.1 Positives

- The device remains responsive while the task is being carried out.
- This pattern utilizes active objects which are significantly more lightweight than threads, with much less RAM overhead. For example, in Symbian OS the absolute minimum overhead for a new thread is 9KB of RAM, approximately 20 times the overhead of introducing an active object where the main RAM cost is the additional code size of roughly 500 bytes.<sup>1</sup>
- In most cases, active objects are as effective as multithreading but are easier to debug, which reduces your maintenance and testing costs. This is not to say that debugging them is always easy, as stray signals can be tricky to track down, however the problems introduced by multithreading are usually more complex and harder to identify.

### 2.3.4.2 Negatives

- The code to carry out the task in stages is more complex than the code would be to carry out the task in a single pass.
- The function called from `ClIdle` is not allowed to leave. This is acceptable if the steps do not themselves call leaving functions. As many functions in Symbian OS can leave, it is likely that some form of TRAP will be necessary which can make the code a little less maintainable.

---

<sup>1</sup> This is the average code size for a class within Symbian OS itself.

- The function called from `CIIdle` is synchronous. This is acceptable if the steps do not themselves make asynchronous calls. If the steps require calls to asynchronous APIs, then this pattern is unlikely to be suitable and using the Asynchronous Controller<sup>2</sup> pattern instead would probably be better.
- The amount of work done in each stage of the task will need to be configured if your code is being run on devices with significantly different processor speeds. For instance, if the stages are implemented so that they take a little under 100ms on an ARMv6 device and then are run on a slower ARMv5 device, the responsiveness of your component will be compromised. However, if you tune the stages such that they take a little under 100ms on an ARMv5 device you will be missing out on efficiencies for the ARMv6 device.

### 2.3.5 Example Resolved

In this section, we take the example task of importing contacts from a file and splitting it up into stages. In this case, we will use both a sequence of stages and an iterative part.

The sequential operations are:

1. read the supplied file into a buffer
2. import the contacts from the buffer.

The final stage of importing contacts entries is iterative, in that it is not necessary to import all the contacts in one pass.

In order to manage the sequence of states, we define an enum type (internally to the servant class) to represent the current state, including an inactive state.

```
private:
    enum TImportState // State of the import task
    {
        EReadingFile,
        EImporting,
        EInactive
    };
```

The servant then needs to include:

- the current state
- a static callback function and a member function called from the static function
- a variety of member variables to persist data between calls to the callback function.

```
class CContactImporter : public CBase
{
public:
    static CContactImporter* NewL();
    ~CContactImporter();

    void ImportL(CContactDatabase* aDB, const TDesC& aFilename);

    static TInt Import(TAny* aArg); // Called by CIIdle
```

---

<sup>2</sup> See the Symbian Press book, *Common Design Patterns for Symbian OS*. More details can be found at [developer.symbian.com/main/documentation/books/books\\_files/pattern](http://developer.symbian.com/main/documentation/books/books_files/pattern).

```

private:
    CContactImporter();
    void ConstructL();
    void CleanUp();

    TInt DoImportL(); // Called from the static Import() function

private:
    TImportState iState;
    CContactDatabase* iCdb;
    TFileName iFileName;
    CBufBase* iBuffer;
    RBufReadStream iReadStream;
    CIIdle* iIdle;
};

```

The constructor and destructor are not unusual in any way and so they are not shown here for the sake of brevity.

Because the class includes a number of member variables to persist state, it needs to be able to clean up and free up memory on completion or cancellation. Note that the task can be cancelled at any point so it is worthwhile creating a clean up method that can be called from anywhere.

```

void CContactImporter::CleanUp()
{
    if(iBuffer)
    {
        iReadStream.Close();
        delete iBuffer;
        iBuffer = NULL;
    }
    if(iIdle)
    {
        delete iIdle;
        iIdle = NULL;
    }
}

```

The details of the member variables cleaned up here will become clearer as the callback function is understood.

The servant provides a member function that is used by the client to prime it with all the data required for the task which, in this case, is a handle to the Contacts database, and the name of the file containing the vCards. As well as storing these variables, the method creates and starts the CIIdle object passing in a TCallback object. The CIIdle object will then take care of calling the callback function repeatedly until told to stop.

```

void CContactImporter::ImportL(CContactDatabase* aCdb, const TDesC& aFileName)
{
    if(iState != EInactive)
    {
        User::Leave(KErrNotReady);
    }
    iState = EReadingFile;
    iCdb = aCdb;
}

```

```

iFileName = aFileName;
TCallback callback = TCallback(Import, (TAny*)this);
idle = CIdle::NewL(CActive::EPriorityIdle);
idle->Start(callback);
}

```

As `CIdle` expects a static, non-leaving function, we have to wrap our leaving member function within a TRAP harness. Note that the `CIdle` object expects the static callback function to return true if work should be continued or false if the task is complete.

```

/* static */ TInt CContactImporter::Import(TAny* aArg)
{
    CContactImporter* importer = (CContactImporter*)aArg;
    TInt impOk = 0;
    TRAPD(err, impOk = importer->DoImportL());
    if (err)
    {
        Cleanup();
    }
    return ((err == KErrNone) && impOk);
}

```

In this case, we stop the task if the member function reports completion by returning zero if it leaves. This is easier than trapping every leaving function within the member function. This frees up the `DoImportL()` function to use the internal state variables in the servant class to choose what work to do for each stage.

```

TInt CContactImporter::DoImportL()
{
    switch(iState)
    {

```

The first stage is to open the supplied file and read the contents into a buffer suitable for importing. This is a simple sequence of synchronous operations (the same as in the synchronous example above). As it uses a loop it could be split up further but we have chosen not to do so in this case. After the file has been read in the internal state is moved on.

```

case(EReadinFile):
{
    // Make a read stream based on a buffer with data read from file
    RFs rfs;
    User::LeaveIfError(rfs.Connect());
    CleanupClosePushL(rfs);
    RFile inFile;
    TInt err = inFile.Open(rfs, iFileName, EFileShareReadersOnly);
    User::LeaveIfError(err);
    CleanupClosePushL(inFile);

    iBuffer = CBufFlat::NewL(KReadSize);
    TBuf8<KReadSize> fileDes;
    do
    {
        err = inFile.Read(fileDes, fileDes.MaxLength());
        if((err == KErrNone) && (fileDes.Length() > 0))
        {
            iBuffer->InsertL(iBuffer->Size(), fileDes);

```

```

    }
} while((err == KErrNone) && (fileDes.Length() > 0));

CleanupStack::PopAndDestroy(2); // inFile, rfs

iReadStream.Open(*iBuffer);

iState = EImporting;
break;
}

```

The second and final stage is to import the contacts from the buffer and is called until all the contacts have been imported. To do this, we make use of the `ImportContactsL()` method provided by the Contacts model, which allows us to import only one entry at a time.

```

case(EImporting):
{
// Import one contact using the data read from the file
TBool importOk = EFalse;
CArrayPtr<CContactItem>* entryArray = iCdb->ImportContactsL(
    TUid::Uid(KUIdVCardConvDefaultImpl),
    iReadStream, importOk,
    CContactDatabase::EImportSingleContact);
delete entryArray; // We don't check what has been imported
if(!importOk)
{ // We've finished so ...
Cleanup();
}
break;
}

```

There are three points to note in the above code fragment. First, it has to detect completion and set the object back to the inactive state. Second, this implementation only imports exactly one entry at a time, which is probably too conservative and therefore inefficient. A better implementation would import a number of entries each time (and the number could be configurable to allow for tuning). In this case, we only report completion by re-setting our internal state. An alternative would be to have an event mixin<sup>3</sup> class observing the task and perhaps being given detailed progress and status information.

Finally, we need a standard default case for the switch statement. We can also include the handling of the inactive state here for reasons of robustness. `CIdle` expects the callback function to return true if there are more stages needed to complete the task or false if it is completed.

```

case(EInactive):
// Unexpected state – drop through to default
default:
{ // Unexpected state
User::Leave(KErrNotReady);
}
} // end switch
return (iState != EInactive);
}

```

---

<sup>3</sup> See the *Common Design Patterns for Symbian OS* book, by Symbian Press for more details on this pattern ([developer.symbian.com/main/documentation/books/books\\_files/pattern](http://developer.symbian.com/main/documentation/books/books_files/pattern)).

## 2.4 Other Known Uses

This pattern is used in a number of places where tasks have an indefinite duration and can be run as 'background' tasks. The Contacts model uses `CIIdle` to sort and compact databases; the sheet engine uses `CIIdle` to recalculate cell values; the messaging server uses `CIIdle` to carry out a range of background operations.

This pattern is also used to set a task to be carried out at some later stage when the phone is idle. The window server uses `CIIdle` to flush client operations; the LBS subsystem uses `CIIdle` to defer the storing of the last known position.

## 2.5 Variants and Extensions

None known.

## 2.6 References

- See the Cooperative Multitasking chapter of the *Common Design Patterns for Symbian OS* book, by Symbian Press, for a description of the Active Object pattern used here as well as the Asynchronous Controller pattern that describes how to solve this problem when `CIIdle` by itself isn't enough, for example.
- You can give feedback or even contribute your own design patterns for Symbian OS on the following wiki page:  
[developer.symbian.com/wiki/display/pub/Common+Design+Patterns+for+Symbian+OS](http://developer.symbian.com/wiki/display/pub/Common+Design+Patterns+for+Symbian+OS).
- See the Developer Library ([developer.symbian.com/main/documentation/sdl](http://developer.symbian.com/main/documentation/sdl)) for more information on the `CIIdle` and `TCallBack` classes as well as the other types used in the example given here such as Contacts database.

## 3 Conclusion

If you like what you read here or you're asking yourself 'How do the experts architect software for mobile devices?' then there is a book 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 for you to 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.

All the patterns in the *Design Patterns* book are tailored specifically for those working with Symbian OS. There are numerous examples provided which demonstrate how each of the patterns work which are implemented in Symbian C++ to help you adapt them for your own software.

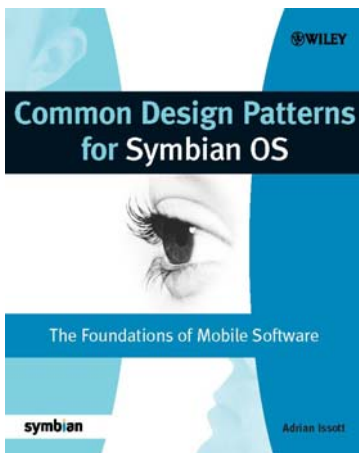
Inside you'll find patterns that illustrate:

- effective error handling
- techniques for working effectively with the constrained resources available to a Symbian smartphone
- event-driven programming to conserve power consumption
- how to take advantage of Symbian's cooperative multitasking framework

- how to provide services to multiple clients, either individually or concurrently
- how to use the platform security architecture to secure your own application and services
- how to optimize execution speeds and start-up times
- the operation of well-known design patterns on Symbian OS, such as Adapter, Singleton and Model-View-Controller.

Whether you are a device creator or an application developer, you will find these patterns help you to write better software that more effectively harnesses the unique characteristics of Symbian smartphones.

## 4 Author Profile



Ian joined Symbian in 2001, has worked in a range of teams within Symbian and is currently a Technology Architect responsible for Shortlink technologies. He has previously filled roles ranging from developer through to project manager to technical manager.

He has an MA in Computer Sciences from Cambridge University and an MBA from Warwick University. As a software engineer for over twenty five years he has worked for a number of software companies and has worked on more than fifteen operating systems, developing software ranging from enterprise systems to embedded software. He is married to Lorraine and they have two children, Kelly and Ross, and a number of pets.

Ian recently contributed two design patterns to the forthcoming Symbian Press book, *Common Design Patterns for Symbian OS*, which will be published in October 2008. More information about this book can be found at [developer.symbian.com/designpatternsbook](http://developer.symbian.com/designpatternsbook).