

# Separated Client-Server: A Design Pattern

John Roe

Published by the Symbian Developer Network

Version: 1.0 – October 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 .....	5
	2.4 OTHER KNOWN USES .....	9
	2.5 VARIANTS AND EXTENSIONS .....	9
	2.6 REFERENCES .....	10
<b>3</b>	<b>CONCLUSION</b> .....	<b>11</b>

# 1 Introduction

This article describes how best to package the well-known Symbian OS Client–Server pattern to maximize encapsulation of the service interface, reduce client dependencies and reduce the overall RAM used by the service. 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*. You can find more information about the book 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 so as 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/fundamentals](http://developer.symbian.com/fundamentals). 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) as well as Symbian Press publications listed at [developer.symbian.com/books](http://developer.symbian.com/books)).

## 2 The Design Pattern

### 2.1 Intent

Implement the client and server components of the Client–Server pattern in different executables to maximize encapsulation, reduce dependencies and minimize RAM consumption.

### 2.2 The Problem

#### 2.2.1 Context

You have chosen to use the Client–Server pattern to synchronize and securely police access to a shared resource or service from multiple client processes.

### 2.2.2 Summary

- A service should be well encapsulated and expose the smallest interface that satisfies the requirements of its clients.
- A service should minimize the resources needed by each of its clients to use the service.
- A service should reduce the dependencies it forces on its clients, particularly on non-Execute In Place (non-XIP) devices.<sup>1</sup>

### 2.2.3 Description

In this pattern we assume that you are familiar with the Client–Server pattern<sup>2</sup>, have chosen to apply it to your specific problem and hence need to decide how to package its various components. The main components that need packaging are:

- The client-side component that exposes a user-friendly interface to the service that hides all the complexity of communicating with a server resident in another process. This is usually a relatively lightweight set of classes that is used by all clients that access the service.
- The server itself and its attendant classes which perform the actions requested of them by their clients. As such, these classes are likely to be significantly more complex than the client-side interface.
- The collection of types shared by both the server and its clients.

When packaging these component parts of the Client–Server pattern, an obvious initial design is to implement the client-side interface and the main server code in the same executable, as this is the simplest arrangement. However, this results in poor encapsulation of the service as any symbols exported by the server will be imported by each client. This will hinder any future extension of the service, as changing these functions would be a compatibility break and hence break existing clients.

However, encapsulation is compromised much more seriously as clients of the service are made statically dependent on all of the server's static dependencies. This is because Symbian OS has a static DLL linkage model such that whenever a DLL is loaded, it is also necessary to load any DLLs which are static dependencies of that DLL. Note that the term 'load' may mean different things according to the memory technology in use in the device. When the DLLs are located on XIP (Execute In Place) media, the load will simply involve creating a handle or reference to the DLL. This is possible because the memory<sup>3</sup> where the DLL's code is located can be directly referenced by the operating system when executing the code and so no extra memory is needed. When DLLs are located on a non-XIP media, such as NAND flash, the operating system needs to physically copy code into RAM before it can be executed. In non-demand-paged systems,<sup>4</sup> the entire DLL will be copied into RAM even if there is code in the DLL that won't be executed

---

<sup>1</sup> A device which is not XIP-based stores code on disk, usually compressed in some way, and loads it into RAM to be run. This is in addition to the RAM used for the runtime data for each instance. Non-XIP devices based on Symbian OS are currently the most common type used outside Japan.

<sup>2</sup> If this is not the case, then please refer to the documents given in the References section below.

<sup>3</sup> Such as NOR flash.

<sup>4</sup> Demand-paging was first introduced as an option in Symbian OS v9.3 and so devices being released at the time of writing will be a mix of demand-paged and non-demand-paged devices. Even then, not all code on a device is necessarily eligible for demand-paging.

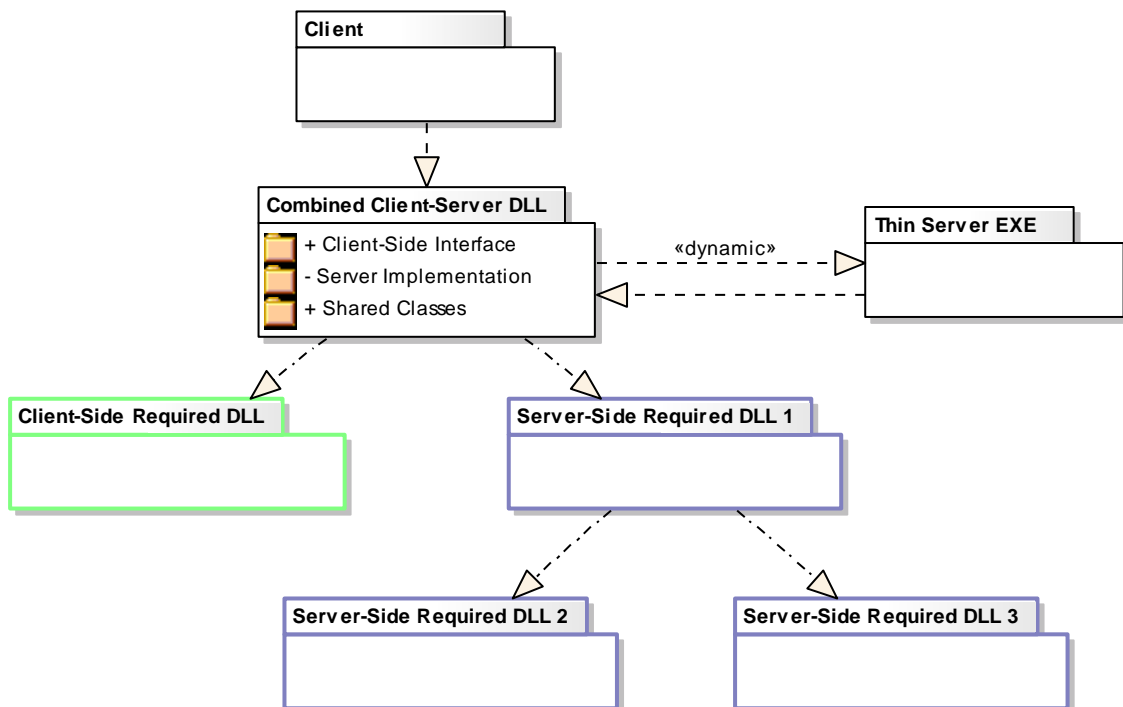
immediately. In a demand-paged system, this is not the case and code is loaded in blocks of 4KB just before it is executed.

The implication of this is that when a DLL, which has a large static dependency tree, is loaded from non-XIP memory in a non-demand-paged system, a significant amount of code needs to be copied into RAM. However, not only is RAM consumed but the process of reading the DLL files from flash and decompressing them can also take a considerable amount of time. Even in demand-paged or XIP devices, an unnecessarily large static dependency tree will be sub-optimal, although it would not be such a critical issue. It is therefore generally desirable to ensure that the minimum number of DLLs are statically loaded for any given use case.

In our situation, where all three major components of the Client–Server pattern are implemented in the same executable, which the majority of clients statically depend on, this means that clients are forced to be dependent on the entire dependency tree of all three components. In particular, when each client starts running, the entire server-side code is loaded even if the client has not yet connected to the server, and hence isn't actually running. On non-XIP devices, this means that the RAM required to hold the server-side code and all its dependencies is used irrespective of whether the server is transient or permanently running.

As a result, attempts to reduce the load on the system by closing servers that are not being used only has a minimal impact, since stopping the server will not actually unload its code from RAM. All that actually happens is that any memory used for the stack or heap(s) of the server will be freed.

Figure 1 below shows how the client is forced to inherit the entire dependency tree of the server if the client and server are implemented in the same DLL.



**Figure 1: Combined Client-Server packaging forces clients to have unnecessary dependencies**

Note that a thin server EXE is still needed to allow a separate process to be created for the server-side to execute within.

## 2.2.4 Example

Consider the central repository, also known as CenRep, which allows data, such as application settings, to be easily persisted across reboots of the device. From the point-of-view of an application using the service to store a configuration setting, the interface is very simple: open a connection to the service and ask for the data to be stored. However, on the server-side, a lot more work is required to satisfy the client's request: authentication of the client to ensure it is allowed to change the repository in question, writing the data to disk in an efficient manner, caching the data to allow it to be quickly accessed in future and many others.

This is reflected in the fact that the CenRep client-side component, and all the code it is statically dependent on, is approximately 260KB in size whilst the server-side component and all its static dependencies are 500KB in size. For code loaded from NAND / non-XIP flash on a device using the multiple memory model<sup>5</sup> and no-demand-paging, this means that the combined packing solution could still take up to 240KB extra RAM, even when the server isn't being used.

## 2.3 The Solution

The solution to packaging the Client–Server pattern is to separate the implementation of the client-side and server components into separate executables. Thus, only when the server is created will its static DLL dependency tree be loaded, and hence only when the server exits will those DLLs be unloaded as well. This will happen independently of the clients, which are then fully decoupled from the server.

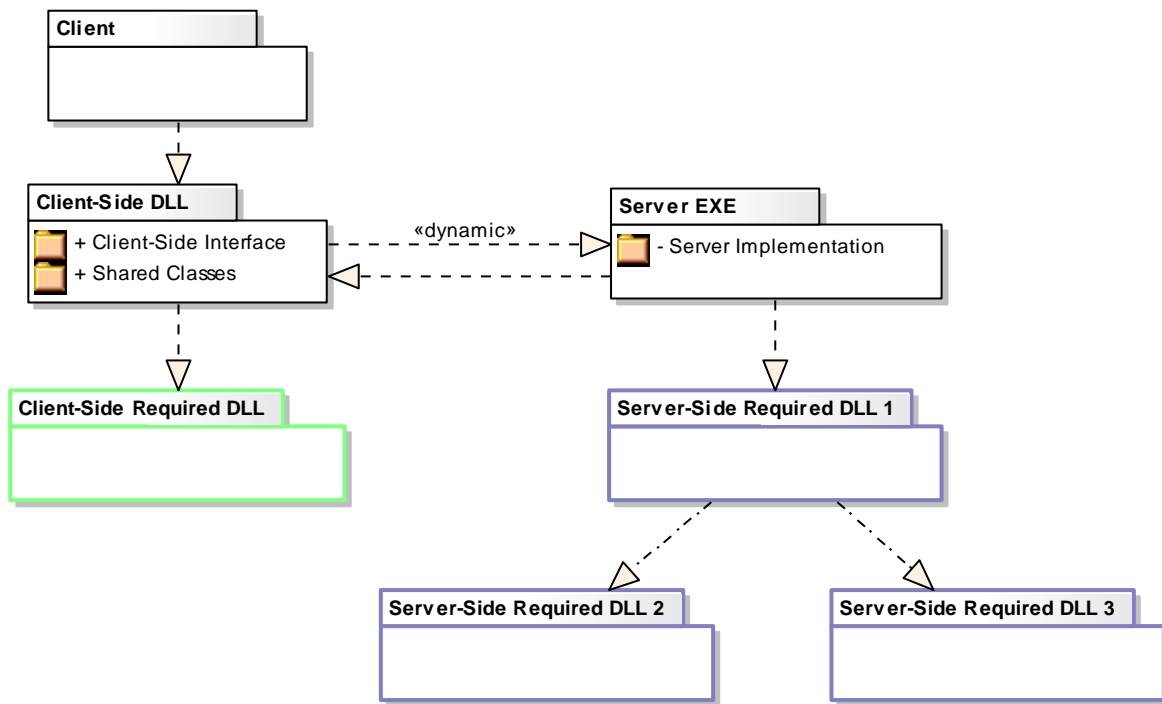
The demand-paging of code addresses a very similar problem to this pattern. Demand-paging is where the operating system only loads executable code into RAM when it is actually executed, after which the code will expire at some future point if it is not used. As a result, demand-paging can prevent the loading of large dependency chains which aren't used, and thus save the memory. Therefore, whilst demand-paging does not conflict with this pattern, it does solve some of the same problems.

### 2.3.1 Structure

Figure 2 shows how clients of the service are isolated from the server's static dependency tree by making them only dynamically dependent on the server.

---

<sup>5</sup> Which memory model is being used is determined by the CPU; ARMv5 usually implies the moving memory model whilst ARMv6 usually implies the multiple memory model. At the time of writing, ARMv6 is the most common type of CPU used on a Symbian OS-based device.



**Figure 2: Separated Client–Server pattern structure**

Note that the diagram above shows the server EXE depending on the client-side DLL. This is because the client-side DLL provides the shared types used in the communication between those two components.

### 2.3.2 Dynamics

By changing the dependency of the client on the server-side from a static to a dynamic dependency, the operating system no longer has to load the server's static dependency tree when the first client is started. Instead, the server's static dependency tree is loaded only when the server is first started, which is usually when the first client *connects* to the service via the client-side interface. This could be some time after the client is loaded and, depending on the use case, may never actually occur if the end user does not activate the particular use case or functionality involving the server. As such, this pattern is best used when clients have a longer lifetime than the service you are designing.

There is a strong relationship with the concept of transient servers in which the Client–Server and Lazy Allocation patterns described in *Common Design Patterns for Symbian OS* are combined. This is where a server is designed to be loaded transiently, such that they only consume system resources when their services are required. However, simply creating a transient server does not in itself save resources. It relies on clients only connecting to the server when they need to use it. If they connect arbitrarily at start-up, then the advantages of the transient server are lost – although of course this is not under the control of the server.

### 2.3.3 Implementation

Implementing this pattern is a simple supplement to the implementation of the Client–Server pattern itself, as it simply involves changing and/or creating MMP files to separate the implementation of the client-side interface from the server implementation. The Symbian Developer Library includes documentation and examples for creating DLLs that you should find useful.

In addition to this, both the client-side DLL and the server EXE should check that they are working with a compatible version of their counterpart. This is because the dynamic nature of the server EXE's dependence on the client-side DLL means that any errors in distributing the two components will only show up at runtime and hence might not be noticed until after they've been released.

In the client-side DLL, the server implementation version required is passed across when the client connects a session using the `RSessionBase`-derived class `RMyClientSession`:

```
EXPORT_C TInt RMyClientSession::Connect()
{
    TInt retry = KMyServerRetryCount;
    for (;;)
    {
        TInt r = CreateSession(KMyServerName,
                               TVersion(KMyVersionMajor,
                                          KMyVersionMinor,
                                          KMyVersionBuild),
                               KMyServerDefaultMessageSlots);
        if (r != KErrNotFound && rKServer != KErrServerTerminated)
        {
            return r;
        }
        if (--retry == 0)
        {
            return r;
        }
        r = StartServer();
        if (r != KErrNone && r != KErrAlreadyExists)
        {
            return r;
        }
    }
}
```

Calling the `CreateSession()` function provided by the client-server architecture on the client-side results in a `NewSessionL()` function being called on the associated `CServer2`-derived class, `CMyServer`. In this function, the server must check that it supports the version of the client-side interface that is trying to connect to it. In the code snippet below, we use the `User` function `QueryVersionSupported()`, as it provides a simple way of specifying that the server supports any client-interface with a major/minor version number less than or equal to the server's version. The expectation here is that the server will maintain backwards compatibility with old interfaces and so we don't insist that the versions match exactly.

```
CMyServerSession2* CMyServer::NewSessionL(const TVersion& aVersion,
                                           const RMessage2&) const
{
    TVersion version(KMyVersionMajor, KMyVersionMinor, KMyVersionBuild);
    if (User::QueryVersionSupported(version, aVersion) == EFalse)
    {
        User::Leave(KErrNotSupported);
    }

    return new(ELeave) CMyServerSession();
}
```

## 2.3.4 Consequences

### 2.3.4.1 Positives

- Good encapsulation of the whole service since separating the client and server components enforces good, private interface design of the communication channel between client and server, especially as there can be no accidental dependency leakage.
- Minimization of the RAM required when clients are running but not using the service. This is felt most on non-XIP systems, where code must be copied into RAM before execution. XIP systems will not benefit significantly as the DLLs are not copied into RAM. However, this is a significant benefit as non-XIP systems are currently the most common type.
- Enhances RAM optimization techniques such as transient use of servers in which the server implementation is only loaded when there are clients connected to the service. If this pattern is not used then a server can still be made transient; but the benefits of doing so will be relatively small, as the executable footprint of the server's dependencies will already be in RAM and only the RAM used for the dynamic footprint (for example, stack and heap) of the server will be optimizable. It is possible that the executable footprint is much larger than the dynamic footprint, and therefore the impact of neglecting to separate the client-side interface and the server code in a transient server could wipe out most of the intended savings.
- Improved client startup time, as the code required by the server will only be loaded when the server is started, rather than when the client starts.

### 2.3.4.2 Negatives

- Sharing code between client and server becomes non-trivial where this is needed.
- The separation of the client-side interface from the server implementation means there is the opportunity for different versions of the two to be used. Hence it is possible to have version X of the client-side DLL and version Y of the server EXE on the same device. This problem can be guarded against by checking versions at runtime at the client-server boundary. However, care must still be taken to ensure that all corresponding versions of the client-side interface and the server are always delivered together.
- Increases the number of small DLLs which is undesirable as more code is required due to the overhead of each DLL. Not only does each DLL have its own header but the representation of each DLL in RAM is aligned to a page<sup>6</sup> boundary – so the more DLLs there are, the more RAM that will be lost rounding up the code size to the next page boundary.<sup>7</sup>
- In situations where all client lifetimes are comparable with the lifetime of the server, this pattern won't actually save any RAM but will instead only add complexity to your design.
- Decreased time taken to first use the server. If the client opens and closes sessions to the server, this could add up to more time wasted than saved in the improved startup time for clients. However, this can be addressed by applying the Lazy De-allocation pattern found in the Resource Lifetimes chapter of *Common Design Patterns for Symbian OS*.

---

<sup>6</sup> Currently 4KB

<sup>7</sup> Executables built into the core image when a non-XIP device is created are an exception to this as they are not page-aligned.

### 2.3.5 Example Resolved

From the previous example, the CenRep client-side implementation is provided in a simple client DLL called `central repository.dll`, which is approximately 3KB in size and only depends on `euser.dll`. Hence this DLL can be used by clients with very little overhead.

The server-side is implemented separately within `central repositorysrv.exe`, which is approximately 35KB in size and has a large dependency tree, as it depends not only on `euser.dll` but also the following executables:

- `efsrv.dll` for file system support
- `estor.dll` for persistent data services
- `charconv.dll` for character encoding and conversion
- `abclient.dll` for active backup support
- `bafl.dll`, `bsulini file.dll` and `sysutil.dll` for application utilities.

By using this pattern, CenRep avoids a static dependency between the client-side interface and the server so that clients are not forced to have unnecessary dependencies on the above executables.

The client-side interface to CenRep only involves the standard Symbian OS types provided by `euser.dll`, so there's no need for `central repositorysrv.exe` to statically depend on `central repository.dll`. Instead, only simple types need to be shared and are provided through header files such as `epoc32\include\central repository.h`.

## 2.4 Other Known Uses

This pattern is used extensively throughout Symbian OS, so we give just two examples here:

- *Contacts Model*

This is a service provided by Symbian OS to allow contacts to be stored in a consistent manner across all Symbian devices. The service is implemented using the client-server pattern. A non-transient server is used as there is no need for the contacts server to always be running and waste RAM. To get the most of this RAM-saving, the Contacts Model makes use of the pattern described.

- *Window Server*

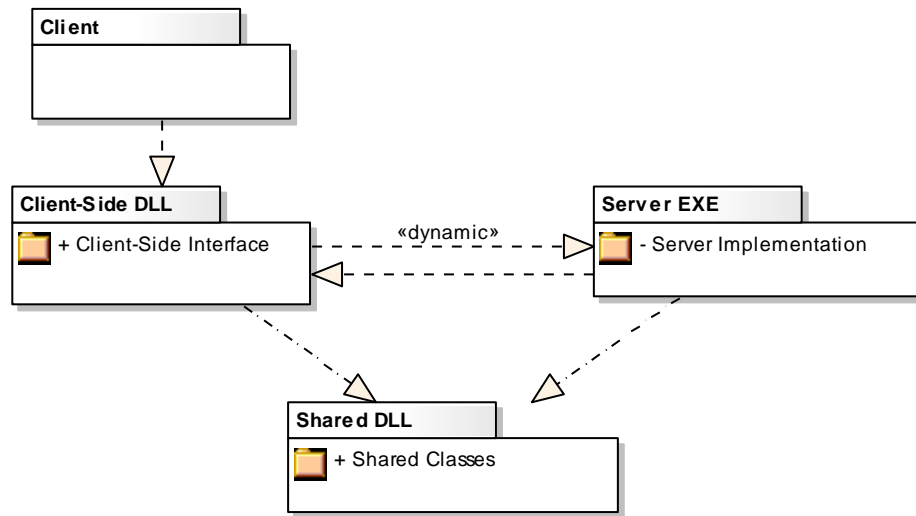
This service is packaged as a separated client-server, with the client-side interface supplied by `ws32.dll` and the server implementation by `ewsrv.exe`. This is despite the fact that the windows server is a non-transient server and hence doesn't gain any RAM benefits from using this pattern. Instead, this pattern is used for the way it increases the encapsulation of the service.

## 2.5 Variants and Extensions

- *Separate Internally Shared Classes*

In some situations, code needs to be shared between the client-side and the server side. One example of this are utility classes that handle the communication channel between these two components. One way to solve this would be to have the client-side DLL implement these classes and export the necessary functions. However, if these classes are only needed internally by the service, and not by its clients, then this isn't very well

encapsulated. The solution is therefore to have an additional DLL that provides those shared classes to both the client-side DLL and the server EXE without them being exposed to clients of the service. This structure is shown in Figure 3.



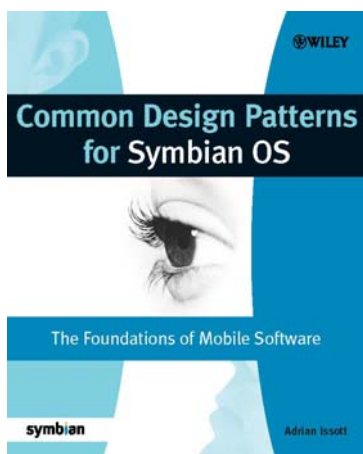
**Figure 3: Separated internally shared classes structure**

## 2.6 References

- Issott, A. et al. (2008) *Common Design Patterns for Symbian OS*, Symbian Press and John Wiley & Sons Ltd.  
Describes a number of design patterns referred to in this document including Client–Server and Lazy De-allocation amongst others.
- Harrison, R. and Shackman, M. (2007) *Symbian OS C++ for Mobile Phones, Volume 3*, Symbian Press and John Wiley & Sons Ltd.  
Contains a chapter on the client–server framework and comprehensively covers how the pattern is codified in Symbian OS.
- Stichbury, J. (2004) *Symbian OS Explained: Effective C++ Programming for Smartphones*, Symbian Press and John Wiley & Sons Ltd.  
Also covers the client–server framework.
- Heath, C. (2006) *Symbian OS Platform Security: Software development using the Symbian OS Security Architecture*, Symbian Press and John Wiley & Sons Ltd.  
Contains a chapter describing how to design a secure server.
- Template code for a transient server is available from [developer.symbian.com/main/documentation/symbian\\_cpp/symbian\\_cpp/#transient](http://developer.symbian.com/main/documentation/symbian_cpp/symbian_cpp/#transient).
- The Symbian Developer Library ([developer.symbian.com/main/documentation/sdl](http://developer.symbian.com/main/documentation/sdl)) contains a number of useful pages including:
  - Symbian Developer Library > Symbian OS Tools And Utilities > Build tools guide > How to build DLLs.

- Symbian Developer Library > Examples > Symbian OS fundamentals example code > CreateStaticDLL and UseStaticDLL: using a statically linked DLL.
  - Symbian Developer Library > Symbian OS guide > Essential idioms > Exporting and Importing Classes in Symbian OS.
  - Symbian Developer Library > Symbian OS guide > Base > Using User Library (E32) > Inter Process Communication > Client/Server Overview.
  - Symbian Developer Library > Examples > User library example code > Client/Server example code.
- Further information on demand paging can be found at [www.symbian.com/symbianos/demandpaging](http://www.symbian.com/symbianos/demandpaging).

### 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 this book are tailored specifically for those working with Symbian OS. There are numerous examples provided that 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. Further information is available at [developer.symbian.com/designpatternsbook](http://developer.symbian.com/designpatternsbook).