

# CActive and Friends

**Ben Morris**

**Published by the Symbian Developer Network**

**Version: v1.89 – June 2008**

<b>1 INTRODUCTION.....</b>	<b>2</b>
<b>2 BACK TO BASICS: EVENTS, ASYNCHRONICITY, MULTI-TASKING, AND MULTI-THREADING .2</b>	<b>2</b>
<b>3 WHY MULTI-THREADING IS COMPLEX .....</b>	<b>3</b>
<b>4 WHY SYMBIAN OS IS DIFFERENT: ACTIVE OBJECTS .....</b>	<b>4</b>
<b>5 THE ACTIVE OBJECT FRAMEWORK .....</b>	<b>5</b>
5.1 CActive .....	5
5.2 CActiveScheduler .....	5
5.3 TIMERS AND OTHER READY-TO-USE ACTIVE OBJECTS.....	6
5.4 ACTIVE OBJECTS AND THREADS .....	6
5.5 ACTIVE OBJECTS AND PREEMPTION.....	6
<b>6 ACTIVE OBJECTS IN PRACTICE .....</b>	<b>7</b>
<b>7 CHECKLIST.....</b>	<b>10</b>
<b>8 ACTIVE OBJECTS IN REAL LIFE .....</b>	<b>11</b>
<b>9 CONCLUSION .....</b>	<b>12</b>
<b>10 FURTHER READING AND WEB RESOURCES.....</b>	<b>12</b>
10.1 READING.....	12
10.2 WEB RESOURCES.....	13
<b>11 AUTHOR PROFILE .....</b>	<b>13</b>

## 1 Introduction

This is the second in a series of papers which examine the Symbian OS native C++ idioms that developers love to hate. This month I look at active objects, a pattern that pervades Symbian OS. If you've done any Symbian OS programming at all, you've met them, even if you didn't know it. More likely, you did know it. Asynchronous programming techniques can be puzzling to developers brought up in the simpler world of straight-ahead, synchronous program control flow. And for those already practised in asynchronous techniques on other systems, the object oriented style of active objects makes them a little different.

Active objects are well documented, in SDKs, on the many Symbian OS programming sites, and in books (see the follow up references at the end of this paper). But they still remain a hot topic in the Knowledgebase and discussion forums, suggesting that developers new to Symbian OS find them tricky, and that they are still not as well understood as they deserve to be.

This paper sets out to explain why they are used in Symbian OS and why they are preferred to multi-threading, and follows up with just enough skeleton code to get a feel for how to use them in practice. Lastly, it provides a checklist of some basic Do's and Don'ts.

## 2 Back to basics: events, asynchronicity, multi-tasking, and multi-threading

Active objects exist to solve a problem. Interactive applications change the rules of program design. Unlike the classic single-purpose, program-as-filter style of programming, an interactive application behaves much more like a conversation with a user; the application's execution path is driven entirely by its responses to user input. This is the 'event driven' programming style popularised first by Mac OS and later by Microsoft Windows (though in one sense it has been around 'for ever'; any kind of state machine is event driven, and state machines have been known for a long time, the most famous example being the original, abstract Turing machine).

The stereotype of an event driven application is that it spends most of its time waiting for user input, but the point is not how busy a program is, but how its control flow is organised. Instead of starting from `main()`, and running to completion in a predefined sequence of operations, an event driven application is conceptually (and may be literally) a single big `switch` statement, fielding and handling arbitrary events the system passes to it.

The resulting programming style is inherently asynchronous. In order for the application to feel snappy and responsive, user input events or events generated indirectly from them, are despatched rapidly to handling methods which complete asynchronously, i.e. outside of the main program flow; the despatch method itself is synchronous (or should appear so, anyway), returning immediately so that it is able to respond to new user input. Then, when an asynchronous method completes, its results are handled by a completion handler, which generates an appropriate completion event. All events, whether they result from synchronous or asynchronous methods, and whether they originate from user or system actions, end up as inputs to the system in the same event queue.

Asynchronous programming has been around for a long time too, particularly as a solution to the problem of the potentially high response delay time, or latency, of peripheral devices. Peripheral devices also typically interface to the external 'real' world, (which includes users!) and so are much more likely to exhibit indeterministic behaviour; printers get switched off, comms lines go down, phones get left off the hook, users get called away. Asynchronous programming allows higher latency events to be moved out of the main flow of program execution so that the rest of the

system can continue to be responsive. This works on any system that supports concurrency (running multiple processes in parallel, or appearing to by time-slicing CPU time).

Early GUI systems like the original Mac OS and Windows implemented their event driven programming models using non-preemptive process scheduling, or co-operative multitasking as it became known. This was in contrast to the Unix and mainframe approach of preemptive process scheduling. The difference is that in a preemptive system, the currently running process can be swapped out at any time by the operating system scheduler. In a non-preemptive system, the scheduler chooses the process which will run next, but always lets it run to completion. Preemption incurs more overhead, and makes the system more complex. But the non-preemptive approach has the serious drawback that badly behaved applications can break a co-operative system by not yielding to allow other processes to be scheduled.

The evolution towards multi-threaded systems can be seen as an attempt to keep the benefits that concurrency brings, including naturally asynchronous and event driven programming idioms, and fully preemptive systems, without suffering the performance penalty of process-based concurrency.

### 3 Why multi-threading is complex

The big problem with multi-threading is that it is complex for application developers. While multi-threaded programming seems to have become ubiquitous, its arrival as a mainstream practice is relatively recent. Web servers, Linux, and new generations of multi-core processors are three reasons for its current popularity, which has seen it elevated from an arcane art practised by an elite few (systems programmers, particularly in areas like comms), to a standard application programming technique.

There are two things in particular that make multi-threading complex:

- Synchronisation; ensuring that events happen in the right order. This is necessary because the sequence in which threads will complete is indeterminate.
- Shared data and the problem of atomicity; protecting the atomicity of data reads and writes. This is necessary, since reads and writes may be invalidated partway through by a preempting thread rewriting the data.

A multi-threaded system must provide appropriate system primitives to allow programmers to manage thread interactions, for example semaphores, locks, mutexes, and the like. Because programmers have to deal directly with this additional complexity, a whole category of potential programming errors emerges, from forgetting to lock critical sections (which risks one thread interfering with program state that another thread depends on), to thread deadlocks and race conditions (in which thread dependencies lead to program failure).

Suddenly, programmers are forced to worry about things that used to come for free, like the integrity and consistency of the runtime context, and have to explicitly manage them at the application level. Even harder, they have to learn to think about potentially complex interactions within their programs which just do not exist in a single threaded application.

This additional complexity motivates the arguments against multi-threading. The debate is certainly not new, and it is probably not over. In a famous Usenix conference presentation in which he concluded that threads are too hard for most programmers, John Ousterhout, inventor of the Tcl language, summarised the drawbacks as follows: multi-threading is hard to debug, breaks abstraction, and can limit instead of improving performance. Even those on the other side of the argument agree that multi-threading is hard.

## 4 Why Symbian OS is different: Active Objects

Symbian OS is a multi-threaded operating system. In particular, threads are the units of execution which are managed, scheduled, run, and switched by the kernel. Its threading model is fully preemptive. Symbian OS is also a real time operating system (since v9), and its thread latencies are guaranteed to be bounded (specifically they are designed to meet the timing needs of a typical mobile phone signalling stack). But, while other operating systems have increasingly adopted multi-threading as the norm at application level, Symbian OS active objects provide an alternative and much simpler model for managing asynchronous behaviour at application level in an event-driven system. In fact, active objects have proved so effective in practice that at most levels above the kernel, they provide the standard pattern for implementing all asynchronous behaviour, including asynchronous system services.

One consequence has been to eliminate the need for semaphores, locks, and other thread primitives, from most Symbian OS programming, leading to a significant increase in simplicity and reduction of potential programming error.

In the device market that Symbian OS targets, which is essentially a consumer market, robustness is a critical success factor, so that eliminating programming errors is a direct contributor to success. But another important rationale for the design of active objects is CPU and power efficiency; Symbian OS is designed for battery powered devices with (relatively) modest CPUs. Typical desktop operating systems, even when they migrate to mobile, are not.

Switching between active objects running in the same thread (which is the typical use case) avoids the overhead of a thread context switch, so an application which uses active objects instead of threads to manage asynchronicity should be more efficient.

From a design perspective, because Symbian OS was designed from the ground up as an object oriented (OO) system, and written almost entirely in C++, an OO implementation of an asynchronous programming pattern was a natural design choice.

In fact active objects are as good an example as any of the way that OO design can be used to encapsulate and hide complexity; and they are a great example of the way that OO is used in Symbian OS to abstract core operating system concepts. An active object encapsulates the mechanisms for making an asynchronous service request and handling the eventual request completion from a single thread, without blocking until completion, and (from the application programmer's point of view) without having to deal with multi-threadedness. Active objects shield application programmers from the trickiness of multi-threading, by moving complexity out of sight into the OS internals.

Active objects give developers what comes very close to a fire and forget mechanism for handling asynchronous events. Under the hood, things are a bit more complex, but in a typical application the complexity is managed by the system. The main thread of every application is provided by the framework with a dedicated active object scheduler and a dedicated active object thread; all active objects launched from the main thread run in the dedicated active object thread, and are scheduled non-preemptively, in priority order. If the application is single-threaded, then all its active objects behave cooperatively. In the bigger context, they are preemptive with respect to the active objects of any other application (or indeed, system thread). To sum up, active objects provide a powerful and simple pattern for structuring applications in an event driven, GUI-based, application model. It preserves the simplicity of single-threadedness, but still provides the benefits of a responsive, reactive, event-based model, all wrapped up in a convenient OO idiom.

And just to repeat the point, their explicit motivation was to make asynchronous, event-driven programming easier for application developers. By removing complexity, they remove the cause of many programming errors, and therefore improve robustness and reliability.

Of course, when you explicitly need to use multi-threading, all the necessary support is available on the platform through native APIs. In addition, Symbian OS v9 introduces an improved level of POSIX compliance (in the form of P.I.P.S, or POSIX on Symbian OS) to make it easier to port existing multi-threaded code to Symbian OS from other platforms, and also to make Symbian OS easier for developers without Symbian OS C++ experience. As a result, much more multi-threaded code is likely to find its way onto the platform. But still, it is not the native way, and multi-threading should be considered the option of last resort, for all the reasons previously discussed.

## 5 The active object framework

The active object framework is defined in `e32base.h`, which you can find in SDKs. The two principal framework classes are `CActive`, which defines the abstract base class from which all active objects are derived, and `CActiveScheduler`, which defines and implements the active scheduler class. (Typically, applications use `CActiveScheduler` rather than deriving from it.)

### 5.1 CActive

`CActive` derives from `CBase`, the standard Symbian OS base class for heap-safe objects. Every active object is `CActive` derived.

- `CActive` defines pure virtual methods `RunL()` and `DoCancel()` which deriving classes must implement.
- `CActive` defines a default `RunError()` method which is called by the active object framework if `RunL()` leaves. Developers are encouraged to override `RunError()` in order to handle error conditions before they propagate to the active scheduler.
- All active objects are constructed with a priority value, which the scheduler uses to schedule them. `CActive` defines priority values ranging from `EPriorityIdle` (-100), good for objects performing background processing, through `EPriorityLow` (-20) and `EPriorityStandard` (0), good for most active objects, to `EPriorityUserInput` (10), which describes itself, and `EPriorityHigh` (20), which should only be used for high priority active objects like timers, to ensure that they have precedence when the active scheduler chooses the next active object to be run.
- `CActive` includes a `TRequestStatus iStatus` member in which the framework stores the completion code when an active object completes.
- `CActive` implements the concrete methods `SetActive()`, `Cancel()`, and `SetPriority()`,

### 5.2 CActiveScheduler

`CActiveScheduler` derives from `CBase`, the standard Symbian OS base class for heap-safe objects. In a typical GUI application the active scheduler is supplied by the framework. Where it needs to be explicitly created, it can usually be instantiated directly from `CActiveScheduler` without derivation.

- `CActiveScheduler` implements the concrete methods `Install()`, which is used to install an active scheduler in cases where the framework has not already done so, `Start()`, and `Stop()` which are used to start and stop a scheduler; and `Add()`, which are used to register an active object with a scheduler.

- CActiveScheduler provides a default Error() implementation, which is called by the framework in the event of an unhandled leave in an active object's RunL() method.
- CActiveScheduler provides the association between an active object's TRequestStatus iStatus status member and its RunL().
- CActiveScheduler implements the scheduling framework which runs the next ready-to-run active object belonging to that thread in priority order.

### 5.3 Timers and other ready-to-use active objects

As well as defining the base classes from which new active objects can be derived, the active object framework also defines some useful ready-made active objects, including:

- concrete class CIdle, a ready-made low priority active object which can be directly instantiated to perform background processing tasks defined by a callback method you supply it
- abstract class CAsyncOneShot, which you can derive from to create a low priority, ready-made, one-off active object
- various CActive derived timers for derivation or direct instantiation, including CTimer, CDeltaTimer, CHeartBeat, CPeriodic.

CTimer, along with its derived periodic and 'heartbeat' timers, should satisfy most timing needs when programming for Symbian OS.

### 5.4 Active objects and threads

Active objects are typically used in Symbian OS to gain the benefits of asynchronous behaviour while avoiding the complexities of multi-threading. However, there is nothing to stop programmers from mixing the two idioms, active objects and multi-threading, where there is good reason to do so.

It is important therefore to understand that active objects and active schedulers are strictly local to the thread from which they are instantiated, and to which they are installed. Every thread from which an active object is instantiated must have its own active scheduler installed, and all active objects instantiated by a particular thread are managed by that active scheduler. It is a programming error to try to invoke an active object created by and registered on one thread, from another thread.

In particular, where the framework supplies an active scheduler, for example for any GUI application, the scheduler is attached to the main application thread only, and can only be used from that thread. If the application launches other threads that use active objects, then each additional thread must instantiate, install, and start its own active scheduler.

### 5.5 Active objects and preemption

Symbian OS threads are preemptable. Because every active scheduler is thread local, the active objects of one thread can be preempted by the active objects of another thread, or alternatively, can preempt them, just as any other thread can preempt or be preempted.

However, an active scheduler itself does *not* implement preemption. Within a thread therefore it is possible for one badly behaved active object to stop others from running, something it is important to be aware of when designing and implementing active objects.

## 6 Active objects in practice

This paper does not set out to be a complete active object tutorial (there are lots of those already; see the follow up references at the end of this paper). The aim of this section is to provide just enough code to get a feel for how active objects are used in practice, without getting lost in the details.

An active object encapsulates both making a request and handling its completion. It also handles cancellation of the request. This encapsulation allows active objects to provide a fire and forget mechanism for managing asynchronous events.

### 1. Making the actual request:

```
iMyRequestor -> Cancel ();
iMyRequestor -> MakeAsynchRequest ();
```

Here, `iMyRequestor` is an active object of type `CMyRequestor`, the purpose of which is to make an asynchronous call to a system service. In this example, the service request is encapsulated in a method named `MakeAsynchRequest()` which `CMyRequestor` defines and implements. The service could be any of the many system services in Symbian OS which define asynchronous client APIs – in this example it's a file server request, `RFs::Notify()`.

Before making the service request a call is made to `Cancel()`, which is a framework defined method, to cancel any outstanding request. See below for more information.

### 2. Handling the completion of the request:

Completion is handled by the `CMyRequestor::RunL()` method. `RunL()` is declared as pure virtual void in the framework. This is where you act on the result of the asynchronous call you made; the framework calls `RunL()` when the service you requested is ready.

There are two important rules about writing a `RunL()` method:

- o `RunL()` should be short and sharp; it should complete its actions promptly and return, since other active objects managed by the same active scheduler will not be able to run until the `RunL()` completes.
- o If `RunL()` may leave, then your active object implementation should override the default `RunError()` method provided by the framework; the framework runs `RunL()` within a trap harness which will catch the leave, and call the active object's `RunError()` method. The default implementation of `RunError()` results in a panic, which is not something you should expose your end users to; implement an appropriate action, and return `KErrNone`.

It is a very common pattern for a `RunL()` to make a new asynchronous request, as well as handling the completion of the previous request. In other words, step 1 is repeated in the `RunL()` method to provide a repeating, periodic asynchronous service.

### 3. Cancelling a request:

You cancel a request by calling the `CActive::Cancel()` method. If there is an outstanding request for which your active object is waiting then `CActive::Cancel()` will invoke your active object's `DoCancel()` method, declared as pure virtual void in the framework. Your implementation should call the cancel method provided by the asynchronous service you are requesting.

To summarise, the declaration of `CMyRequestor` might look like this:

```

class CMyRequestor : public CActive
{
public:
    // Standard C++ destructor, Symbian OS 2 phase construction
    ~CMyRequestor();
    static CMyRequestor* NewL();
    // The asynch request
    void MakeAsynchRequest();
protected:
    CMyRequestor(); // C++ default constructor
    void ConstructL(); // 2nd phase constructor, may leave
protected:
    // From CActive
    virtual void RunL();
    virtual void DoCancel();
    virtual TInt RunError(TInt aError);
private:
    // Specific to our active object, not related to the framework
    RFs iFs; // File server session handle used to make an
            // asynchronous file server request
    HBufC* iPath; // Argument we give when making the file server
                // request
};

```

An active object implementation will implement the RunL() and DoCancel() pure virtual functions declared by the framework, and should typically also override the framework default RunError() if the RunL() may leave.

A typical RunL() implementation might look like this:

```

void CMyRequestor::RunL()
{
    // If an error occurred, handle it in RunError()
    User::LeaveIfError(iStatus.Int());

    // Make a new asynchronous request immediately if this is a
    // periodic or continuous service
    iFs.NotifyChange(ENotifyAll, iStatus, *iPath);
    SetActive(); // Mark this object as active

    // Now handle the event completion
    ...
}

```

Construction should follow the standard two-phase Symbian OS idiom, with a public NewL() factory method and protected or private C++ constructor and ConstructL() method.

The C++ constructor, by the way, must do two things:

- set the priority of the active object, using initialisation list syntax
- register the active object with the active scheduler, using the CActiveScheduler static method Add().

For example:

```
// Active object C++ constructor
CMyRequestor::CMyRequestor()
    : CActive(CActive::EPriorityStandard)
    {
    CActiveScheduler::Add(this);
    }
```

Finally, of course, the active object must implement its request to the asynchronous service provider, which is after all the reason for the class to exist in the first place. In this example, the service being requested is a file server service which requests a notification of change to the files or directories indicated by the *\*iPath* argument.

```
// Asynchronous service requestor method
CMyRequestor::MakeAsynchRequest()
    {
    // Only one request may be outstanding at any time
    // Cancel() should be called before calling this method

    // Test whether a request is already outstanding
    if (IsActive())
        {
        // This is a programming error, so a panic is appropriate
        _LIT(KMyRequestorPanic, "CMyRequestor");
        User::Panic(KMyRequestorPanic, KErrInUse);
        }

    // Otherwise make the request
    iFs.NotifyChange(ENotifyAll, iStatus, *iPath);
    SetActive(); // Mark this object as active
    }
```

Note that in the code above:

- CActive::IsActive() is a framework method which tests whether there is a request outstanding for which this active object is waiting. If the test fails, raising a panic is appropriate, since this is a programming error. You should always perform this test before requesting the asynchronous service and calling SetActive() on the request. The test could also take the form of an ASSERT statement, in this case:

```
_LIT(KMyRequestorPanic, "CMyRequestor");
__ASSERT_ALWAYS(!IsActive(), User::Panic(KMyRequestorPanic, KErrInUse));
```

- iStatus is a protected TRequestStatus object which the framework declares in the CActive class: the derived active object class therefore does not need to declare it or instantiate it, but just uses it. It is always passed as a parameter in an asynchronous service request, and in fact its presence as a parameter in a service method is the tell-tale sign that the method is asynchronous.
- CActive::SetActive() is the framework method which marks an active object in the active scheduler list of registered active objects as active. You *must* call this method immediately after issuing a request to an asynchronous service.

There is a limit to the value of toy code, and this is not a complete example. Volume 3 of *Symbian OS C++ for Mobile Phones* (Harrison and Shackman, 2007) has several complete active object examples which you can download from the [Symbian Press pages](#) on the Symbian Developer Network. Chapter 6 is a complete and detailed exploration of active objects.

## 7 Checklist

The checklist below summarises some of the more important points about using active objects.

- Applications get an active scheduler for free, provided by the framework to manage any active objects created from the main application thread. But if you create new threads in an application, you will need to instantiate and install an active server for each new thread that creates an active object, and you will need to instantiate and install your own active scheduler if you write a server or console application (a test harness, for example) which uses active objects.

The code to do so is shown below, but you should follow up the references at the end of this paper for a fuller discussion:

```
CActiveScheduler* scheduler = new(ELeave) CActiveScheduler;
CleanupStack::PushL(scheduler);
CActiveScheduler::Install(scheduler);
```

Once installed, the start and stop methods `CActiveScheduler::Start()` and `CActiveScheduler::Stop()` are used to start and stop the scheduler.

- Create your active object and, as part of your default construction, add it to the scheduler using `CActiveScheduler::Add()`. Note that `Add()` is a static method and therefore knows which scheduler to use; you don't need to supply a reference the scheduler.
- In your active object class, you submit an asynchronous request by passing (a reference to) the `iStatus` member of your class. You don't need to declare an additional `TRequestStatus` for the request (a common mistake!), just use the one you are given by the `CActive` base class.
- The asynchronous service provider sets the `iStatus` value to `KRequestPending`, which the framework uses to identify active objects which are waiting on completion. You do not need to do this in your active object code.
- You should test for any outstanding requests by calling `CActive::IsActive()` before making an asynchronous service request. This check *is* important. If an active object succeeds in making a new request before the currently outstanding request has completed, the likely consequence is a stray signal and a system panic.
- Always call `CActive::SetActive()` immediately after making an asynchronous service request.
- Handle completion of the asynchronous request in your `RunL()` method.
- Implement the `RunError()` method to handle errors. Return `KErrNone` for errors you can handle, and actual error codes for any errors you can't. If you don't handle the errors in the active object, the active scheduler will pass control to its `Error()` method, and you may not want that behaviour (for example, if the scheduler raises a panic).

- Cancel an active object using `CActive::Cancel()`. Be sure always to cancel an active object before destroying it. Failing to do so and leaving a request outstanding will cause a stray signal when the active scheduler tries to invoke the object's no longer existing `RunL()`. Therefore you should `CActive::Cancel()` in your destructor. Always call the framework method by the way, not the `DoCancel()` method of your derived active object.
- The `TRequestStatus aStatus` parameter in a method signature always indicates an asynchronous method, which you should call from an active object. Many APIs, by the way, pair asynchronous and synchronous versions of otherwise identical methods (typically the synchronous version is just a wrapped version of the standard asynchronous version).
- If you mix active objects and threads, remember that every active object belongs to an active scheduler, and that active schedulers are thread specific; trying to call methods on an active object running in one thread from a different thread won't work. All steps in making a given asynchronous request must be implemented from the same thread.
- Do not allocate active objects on the stack (i.e. as automatic variables), because completion of the asynchronous service request may occur beyond the scope of an automatic variable.
- Selecting appropriate priorities for your active objects is important, since they will be priority scheduled. In most cases `EPriorityStandard(0)` is appropriate, but to handle direct user input, or the main server object in the case of a server, use `EPriorityUserInput(10)`; long running tasks should use `EPriorityIdle`.
- `RunL()` must always complete in reasonable time, since any other active objects owned by the same active scheduler will be queued, waiting for its completion. Intrinsically long-running completions should be chunked. The Store component, for example, performs compaction incrementally, not monolithically.
- The most common problem encountered when using active objects is a stray signal panic. The most common causes of stray signals when using active objects are:
  - not calling `CActiveScheduler::Add()`, resulting in your active object not being registered
  - not calling `SetActive()` after submitting a request to a service provider, resulting in your request not being registered
  - issuing a new request when a request is already outstanding.

## 8 Active objects in real life

Active objects really are ubiquitous in Symbian OS. The UI Framework, for example, includes two active objects, one handling all user input, the other handling all window redrawing. These underlie such application framework methods as `HandleCommandL()` (in the `CAppUI` class), and `Draw()`, `OfferKeyEventL()`, `HandlePointerEvent()` and others in Control Environment (a.k.a. CONE). Many more work invisibly in the background.

To take another system example, active objects are used within the window server to handle key repeats, and more complex use is made of them in the window server's event handling code. Window server events, in fact, are themselves active objects (of class `CEvent`, which is `CActive` derived).

As a final example of how deeply embedded in the system active objects are, the client server framework implementation is itself based on active objects; server framework classes are active object derived, all the way to their (non-public) `RunL()` and `DoCancel()` methods.

Above all, Symbian OS active objects provide an elegant solution to the problem of making asynchronous programming manageable for application level programming, and for ordinary developers.

## 9 Conclusion

I hope I have shown that active objects are quite straightforward to implement. That does not automatically make them easy to use, but the true complexity is not in the active object idiom itself, but in the underlying problem of managing asynchronicity within programs. Within Symbian OS itself, active objects have proved themselves robust, reliable, and much less prone to error than the alternatives. On that basis, they have become widely used throughout the system.

Others might phrase that more strongly. For example, at one time there was an internal document circulating inside Symbian which opened with these words:

*Are you unsure about active objects, or unconvinced of their virtues? Then clear off and stop reading this document now!*

You get the idea: active objects are held in high regard within Symbian, and they go back a long way. Around 1991 they were first implemented in Psion's 16-bit SIBO system, a precursor operating system to Symbian OS. When work on Symbian OS began, the active object idiom was already tried and tested and proven.

In the wider world, the book *Pattern Languages of Program Design 2* (1996) documented the active object idiom, as did an influential article by Martin Carroll (1998) of Bell Labs.

## 10 Further reading and web resources

### 10.1 Reading

Any of the Symbian Press programming books for C++ developers will give you the active object basics. The following three references are a good place to start:

#### **Richard Harrison and Mark Shackman (2007)**

*Symbian OS C++ for Mobile Phones, Volume 3, Wiley/Symbian Press*

- The latest version of the classic book. Chapter 6 is devoted to active objects.

#### **Jo Stichbury and Mark Jacobs (2006)**

*The Accredited Symbian Developer Primer, Wiley/Symbian Press*

- Chapter 9 introduces the principles of event driven programming, and then explores active objects in some depth.

#### **Steve Babin (2007)**

*Developing Software for Symbian OS, Second Edition, Wiley/Symbian Press*

- Chapter 8 discusses active objects in detail.

For some of the background and history surrounding Symbian OS, including active objects, and for an architectural perspective, see my book:

**Ben Morris (2007)**

*The Symbian OS Architecture Sourcebook*, Wiley/Symbian Press

- Covers the architectural view, as well as some historical context.

Other references are to:

**John Vlissides, Jim Coplien, and Norm Kerth (1996)**

*Pattern Languages of Program Design 2*, Addison-Wesley Longman

- Classic book from the patterns movement; includes an active object pattern.

The Martin Carrol paper is in *Software Practice and Experience* volume 28, issue 1 pp 1-21 (1998)

**10.2 Web resources**

Putting the case against multi-threading is John Ousterhout (1996) *Why threads are a bad idea (for most purposes)*. Invited talk at the Usenix Technical Conference. It can be found at <http://home.pacbell.net/ouster/threads.pdf>.

Herb Sutter in *The Free Lunch is Over* makes the argument in favour of concurrency, without denying that multi-threading is just too hard. It can be found at <http://www.gotw.ca/publications/concurrency-ddj.htm>.

On the web, Forum Nokia provides some useful resources on the Forum Nokia wiki at [http://wiki.forum.nokia.com/index.php/Active\\_object](http://wiki.forum.nokia.com/index.php/Active_object); as well as publishing the [Symbian OS basics workbook](#).

You might also like to look up these two Symbian Developer Network Knowledgebase items, which give a useful insight into what can go wrong when writing active objects; both concern an infamous View Server panic: [FAQ-0900](#) and [FAQ-0920](#).

As always, the Symbian Developer Network is at [developer.symbian.com](http://developer.symbian.com).

**11 Author Profile**

Ben Morris freelances as a writer and software architect specialising in Symbian OS. He is the author of *The Symbian OS Architecture Sourcebook: Design and Evolution of a Mobile Phone OS*, published by Symbian Press. He can be contacted through [www.benmorris.eu](http://www.benmorris.eu).