3

Introduction to the Architecture of Symbian OS

3.1 Design Goals and Architecture

Architecture is goal driven. The architecture of a system is the vehicle through which its design goals are realized. Even systems with relatively little formal architecture, such as Unix,[1] evolve according to more or less well-understood principles, to meet more or less well-understood goals. And while not all systems are 'architected', all systems have an architecture.

Symbian OS follows a small number of strong design principles. Many of these principles evolved as responses to the product ethos that was dominant when the system was first being designed.[2] That ethos can be summarized in a few simple rules.

- User data is sacred.
- User time is precious.
- All resources are scarce.

And perhaps this one too, 'while beauty is in the eye of the beholder, elegance springs from deep within a system'.

In Symbian OS, that mantra is taken seriously. What results is a handful of key design principles:

- ubiquitous use of servers: typically, resources are brokered by servers; since the kernel itself is a server, this includes kernel-owned resources represented by R classes

[1] 'Bottom up' and 'informal' typify the Unix design approach, see [Raymond 2004, p. 11].

[2] That is, the ethos which characterized Psion in the early-to-mid 1990s. By then, the company was a leader in the palmtop computer market. It was a product company.

- pervasive asynchronous services: all resources are available to multiple simultaneous clients; in other words, it is a service request and callback model rather than a blocking model
- rigorous separation of user interfaces from services
- rigorous separation of application user interfaces from engines
- engine reuse and openness of engine APIs.

Two further principles follow from specific product requirements:

- pervasive support for instant availability and instant switching of applications
- always-on systems, capable of running forever: robust management and reclaiming of system resources.

Symbian OS certainly aims at unequaled robustness, making strong guarantees about the integrity and safety (security) of user data and the ability of the system to run without failure (to be crash-proof, in other words). From the beginning, it has also aimed to be easy and intuitive to use and fully driven by a graphical user interface (GUI). (The original conception included a full set of integrated applications and an attractive, intuitive and usable GUI; 'charming the user' is an early Symbian OS slogan.[3])

Perhaps as important as anything else, the operating system set out from the beginning to be extensible, providing open application programming interfaces (APIs), including native APIs as well as support for the Visual Basic-like OPL language and Java, and easy access to Software Development Kits (SDKs)[4] and development tools.

However, systems do not stand still; architectures are dynamic and evolve. Symbian OS has been in a state of continuous evolution since it first reached market in late 2000; and for the three years before that it had been evolving from a PDA operating system to one specifically targeting the emerging market for mobile phones equipped with PDA functions. In view of this, it may seem remarkable that the operating system exhibits as much clarity and consistency in design as it does.

[3] For example, see almost anything written by David Wood. Today, the GUI is no longer supplied by Symbian, however GUI operation remains intrinsic to the system design. The original integrated applications survive in the form of common application engines across multiple GUIs, although their inclusion is a licensee option.

[4] Symbian no longer directly supplies SDKs, since these are GUI-dependent. Symbian provides significant 'precursor' content to licensees for inclusion in SDKs, including the standard documentation set for Symbian OS APIs.

Architectures evolve partly driven by pressures from within the system and partly they evolve under external pressures, such as pressures from the broad market, from customers and from competition.

Recent major releases of Symbian OS have introduced some radical changes, in particular:

- a real-time kernel, driven by evolving future market needs, in particular, phone vendors chasing new designs (for example, 'single core' phones) and new features (for example, multimedia)
- platform security, driven by broader market needs including operator, user and licensee needs for a secure software platform.

While both are significant (and profound) changes, from a system perspective they have had a relatively small impact on the overall shape of the system. Interestingly, in both cases the pressure to address these particular market needs arose internally in Symbian in anticipation of the future market and ahead of demand from customers.

It is tempting to idealize architecture. In the real world, all software architecture is a combination of principle and expediency, purity and pragmatism. Through the system lifecycle, for anything but the shortest-lived systems, it is part genuine, forward-looking design and part retrofitting; in other words, part architecture and part re-architecture.

Some of the patterns that are present in Symbian OS were also present (or, in any case, had been tried out) in its immediate precursors, the earlier Psion operating systems. The 16-bit operating system (SIBO) had extended the basic server-based, asynchronous, multitasking model of previous Psion products and re-engineered it using object-oriented techniques. SIBO also pioneered the approach to GUI design, designed communications services into the system at a deep level, and experimented with some idioms which have since become strongly identified with Symbian OS (active objects, for example).

In fact, surprisingly many features of Symbian OS have evolved from features of the earlier system:

- the fully integrated application suite: even though Symbian OS no longer includes a user interface or applications, it remains strongly application-centric
- ubiquitous asynchronous services
- optimization for battery-based devices
- optimization for a ROM-based design: unlike other common operating systems, SIBO used strategies such as 'execute-in-place' (XIP) (compare this with MS-DOS, which assumes it is loaded into RAM

to execute) and re-entrancy[5] (MS-DOS is non-re-entrant), as well as a design for devices with only solid-state disks

- sophisticated graphical design: from the beginning, SIBO supported reactive repainting of windows and overlapping windows, in an age of tiled interfaces (for example, Windows 2.0 and the character-mode multitasking user interfaces of the day, such as TopView and DesqView)
- an event-driven programming model
- cross-platform development: the developers' mindset was more that of embedded systems engineering than the standard micro-computer or PC model.[6]

SIBO also introduced some of the programming constraints which show up in Symbian OS, for example forbidding global static variables in DLLs (because the compilers of the day could not support re-entrant DLLs), an early example of using the language and tools to constrain developer choices and enforce design and implementation choices, a consistent theme in Symbian's approach to development.

Symbian OS, or EPOC as it was then, was able to benefit from the experience of the earlier implementation in SIBO. The 16-bit system was, in effect, an advanced prototype for EPOC.

Meanwhile, of course, Symbian OS has continued to evolve. In particular, some crucial market assumptions have changed. Symbian OS no longer includes its own GUI, for example; instead it supplies the framework from which custom, product-ready GUIs such as S60, MOAP and UIQ are built. Hardware assumptions have changed quite radically too. Execute-in-place ROMs, for example, depend on byte-addressable flash silicon (so-called NOR flash); more recently, non-byte-addressable NAND flash has almost wholly superseded NOR flash, making execute-in-place a redundant strategy. Other technology areas, for example display technologies, have evolved almost beyond recognition compared to the 4-bit and 8-bit grayscale displays of earlier times. Not least, the telephony standards that drive the market have evolved significantly since the creation of the first mobile phone networks.

Despite sometimes radical re-invention and change, the original design conception of Symbian OS is remarkably intact.

---

[5] In designing for re-entrant DLLs (that is, re-entrant shared libraries), SIBO was significantly in advance of the available tools. For example, C compilers were poor in this area. Geert Bollen makes the point that it is not just language features that determine whether a given language is suitable for a particular project; the tools infrastructure that supports the language is equally important.

[6] It is interesting to note that Bill Gates has identified as one of Microsoft's key strengths (and, indeed, a key competitive advantage), that it develops all of its systems on its own systems. The advantage breaks down completely in the mobile phone context.

3.2 Basic Design Patterns of Symbian OS

The design principles of a system derive from its design goals and are realized in the concrete design patterns of the system. The key design patterns of Symbian OS include the following:

- the microkernel pattern: kernel responsibilities are reduced to an essential minimum
- the client–server pattern: resources are shared between multiple users, whether system services or applications
- frameworks: design patterns are used at all levels, from applications (plug-ins to the application framework) to device drivers (plug-ins to the kernel-side device-driver framework) and at all levels in between, but especially for hardware adaptation-level interfaces
- the graphical application model: all applications are GUI and only servers have no user interface
- an event-based application model: all user interaction is captured as events that are made available to applications through the event queue
- specific idioms aimed at improving robustness: for example, active objects manage asynchronous services (in preference, for example, to explicit multi-threading) and descriptors are used for type-safe and memory-safe strings
- streams and stores for persistent data storage: the natural 'document' model for applications (although conventional file-based application idioms are supported)
- the class library (the User Library) providing other user services and access to kernel services.

3.3 Why Architecture Matters

'Doing architecture' in a complex commercial context is not easy. Arguably all commercial contexts are complex (certainly they are all different), in which case architecture will never be easy. However, the business model for Symbian OS is particularly complex. While it must be counted as part of Symbian's success, it also creates a unique set of problems to overcome and work around, and to some extent those problems are then manifested as problems for software architecture.

Architecture serves a concrete purpose; it makes management of the system easier or more difficult, in particular:

- managing the functional behavior and supported technologies
- managing the size and performance
- retaining the ability to evolve the system.

Elegance, consistency, and transparency were all early design drivers in the creation of the system. Charles Davies, now Symbian CTO, was the early architect of the higher layers of the operating system.

Charles Davies:

I remember looking at Windows at that time and thinking that this is all very well, here is this Windows API, but just look what's happening underneath it, it was ugly. I wanted something that you could look inside of.

The early 'ethic of robustness', to use his phrase, came straight from the product vision.

Managing the Bounds of the System

In some ways, the hardest thing of all for Symbian is managing the impact of its business model on the properties of the system and, in particular, the problem that Charles Davies calls 'defining the skin' – understanding, maintaining, and managing the bounds of the system under the impact of the business model. As well as originating the requirements push and feeding the requirements pipeline, generating almost all of the external pressure on the system to evolve and grow, licensees and partners also create their own extensions to the system. (S60, arguably, is the most extreme example, constituting a complete system in itself, at around twice the size of the operating system.)

Being clear where to draw the boundary between the responsibilities of Symbian OS and the responsibilities of GUIs, in terms of who makes what and where the results fit into the architecture, becomes difficult. Charles Davies is eloquent on the subject.

Charles Davies:

One of the things I've done since being here is to try and identify where the skin of Symbian OS is, to define the skin. When I was at Psion and we were building a PDA, I understood where the PDA ended and where the things outside the PDA began, and so I knew the boundaries of the product. And then I came to Symbian and Symbian OS, and I thought, where are the boundaries? It's really tough to know where the boundaries are, and I still sometimes wonder if we really know that. That's debilitating from the point of view of knowing what to do. In reality we're trying to fit some kind of rational

> boundary to our throughput, because you can't do everything. We've got, say, 750 people in software engineering working on Symbian OS, and we can't make that 1500 and we don't want to make that 200. So with 750 people, what boundary can we draw that matches a decent product?

In one sense the problem is particular to the business model that Symbian has evolved, and is less a question of pure technology management, which to some extent takes care of itself (or should, with a little help to balance the sometimes competing needs of different licensees), than of driving the operating system vision in the absence of a wider product vision. In that wider sense, the licensees have products; Symbian OS has technologies and it is harder to say what the source of the technology vision for the operating system itself should be. To remain at the front of the field, Symbian OS must lead, but on the whole, customers would prefer that the operating system simply maps the needs of their own leading products. The problem is that by the time the customer product need emerges, the operating system is going to be too late if it cannot already support it. (At least, in the case of complex technologies and, increasingly, all new mobile technologies are complex.) Customers therefore build their own extensions or license them from elsewhere, and the operating system potentially fails under the weight of incompatibilities or missing technologies).

Product companies are easier to manage than technology companies because it is clear what product needs are; either they align with the market needs or the product fails in the market. The Symbian model is harder and forever raises the question of whether Symbian is simply a supplier or integrator to its customers, or an innovator. Is Symbian a product company, where the operating system is the product, or does it merely provide a useful 'bag of bits'?

Architecture is at the heart of the answer. If there is an architecture to describe, then there is more to the operating system than the sum of its parts.

Managing Competitive Threats

There are many external threats to Symbian OS. Some of the threats are household names. Microsoft is an obvious threat, but the likelihood is that Microsoft itself will always be unacceptable to some part of the market, whatever the quality of its technology offering. (It is hard to see Nokia phones, for example, sharing branding with Microsoft Windows, and the same issues no doubt apply to some network operators, but clearly not to all of them.) It is almost as certain that Nokia in turn is unacceptable to some other parts of the market. S60 aims at building a stable of licensees, vendors for whom the advantages of adopting a proven, market-ready

user interface outweigh the possible disadvantages of licensing a solution from a competitor, or the costs of developing an in-house solution. There will always be vendors, though, for whom licensing from Nokia is likely to be unacceptable. Interestingly, the more Microsoft resorts to branding its own phones, in order to increase market share, the more it competes with those it is seeking to license to. It is hard to see any scenario in which the phone market could become as homogeneous as the PC market.

Linux is also a clear and visible threat, even though again there are natural pockets of resistance. Linux, for example, is viral. Linux does not just take out competitors, it takes out whole parts of the software economy, and it is not yet clear what it replaces them with.[7] To put Linux in a phone, for example, seems to require just the same 'old' software economy as to put any other operating system into a phone, dedicated software development divisions which do the same things that other software development divisions do: write code, miss deadlines, fix defects, pay salaries. Linux may be royalty-free, but that translates into 'not-free-at-all' if you have to bring it inside your own dedicated software division. Nonetheless, to ignore Linux would be a (possibly fatal) mistake.

Architecture is part of the answer. If Symbian OS is a better solution, it is because its architecture is more fit for purpose than that of its competitors, not because its implementation is better. Implementation is a second-order property, easy to replace or improve. Architecture, in contrast, is a deep property.

3.4 Symbian OS Layer by Layer

The simplest architectural view of Symbian OS is the layered view given by the Symbian OS System Model.[8]

UI Framework Layer

The topmost layer of Symbian OS, the UI Framework layer provides the frameworks and libraries for constructing a user interface, including the basic class hierarchies for user interface controls and other frameworks and utilities used by user interface components.

The UI Framework layer also includes a number of specialist, graphics-based frameworks which are used by the user interface but which are also available to applications, including the Animation framework, the Front End Processor (FEP) base framework and Grid.

The user interface architecture in Symbian OS is based on a core framework called Uikon and a class hierarchy for user interface controls

[7] Where it is clear, it is not clear how to make a profit from what it replaces them with.

[8] The System Model (see Chapter 5) is relatively constant across different releases, although its details evolve to track the evolution of the architecture.

called the control environment. Together, they provide the framework which defines basic GUI behavior, which is specialized by a concrete GUI implementation (for example, S60, UIQ or MOAP), and the internal plumbing which integrates the GUI with the underlying graphics architecture.

Uikon was originally created as a refactoring of the Eikon user interface library, which was part of the earliest versions of the operating system. Uikon was created to support easier user interface customization, including 'pluggable' look-and-feel modules.

The Application Services Layer

The Application Services layer provides support independent of the user interface for applications on Symbian OS. These services divide into three broad groupings:

- system-level services used by all applications, for example the Application Architecture or Text Handling
- services that support generic types of application and application-like services, for example personal productivity applications (vCard and vCal, Alarm Server) and data synchronization services (OMA Data Sync, for example); also included are a number of key application engines which are used and extended by licensees (Calendar and Agenda Model), as well as legacy engines which licensees may choose to retain (Data Engine)
- services based on more generic but application-centric technologies, for example mail, messaging and browsing (Messaging Store, MIME Recognition Framework, HTTP Transport Framework).

Applications in Symbian OS broadly follow the classic object-oriented Model–Viewer–Controller (MVC) pattern. The framework level support encapsulates the essential relationships between the main application classes (representing the application data model, the views onto it, and the document and document user interface that allow it to be manipulated and persisted) and abstracts all of the necessary underlying system-level behavior. In principle, a complete application can be written without any further direct dependencies (with the exception of the User Library).

The Application Services layer reflects the way that the system as a whole has evolved. On the one hand, it contains essential application engines that almost no device can do without (the Contacts Model for example), as well as a small number of application engines that are mostly now considered legacy (e.g. the WYSIWYG printing services and the office application engines, including Sheet Engine, a full spreadsheet engine more appropriate for PDA-style devices). On the other hand, it contains (from Symbian OS v9.3) the SIP Framework, which provides the foundation for the next generation of mobile applications and services.

Java ME

In some senses, Java does not fit neatly into the layered operating system model. Symbian's Java implementation is based around:

- a virtual machine (VM) and layered support for the Java system which complements it, based on the MIDP 2.0 Profile
- a set of standard MIDP 2.0 Packages
- an implementation of the CLDC 1.1 language, I/O, and utilities services
- a number of low-level plug-ins which implement the interface between CLDC, the supported packages, and the native system.

Java support has been included in Symbian OS from the beginning, but the early Java system was based on pJava and JavaPhone. A standard system based on Java ME first appeared in Symbian OS v7.0s. Since Symbian OS v8, the Java VM has been a port of Sun's CLDC HI.

The OS Services Layer

The OS Services layer is, in effect, the 'middleware' layer of Symbian OS, providing the servers, frameworks, and libraries that extend the bare system below it into a complete operating system.

The services are divided into four major blocks, by broad functional area:

- generic operating system services
- communications services
- multimedia and graphics services
- connectivity services.

Together, these provide technology-specific but application-independent services in the operating system. In particular, the following servers are found here:

- communications framework: the Comms Root Server and ESock (Sockets) Server provide the foundation for all communications services
- telephony: ETel (Telephony) Server, Fax Server and the principal servers for all telephony-based services
- networking: the TCP/IPv4/v6 networking stack implementation
- serial communications: the C32 (Serial) Server, providing standard serial communications support

- graphics and event handling: the Window Server and Font and Bitmap Server provide all screen-drawing and font support, as well as system- and application-event handling
- connectivity: the Software Install Server, Remote File Server and Secure Backup Socket Server provide the foundation for connectivity services
- generic: the Task Scheduler provides scheduled task launching.

Among the other important frameworks and libraries found in this layer is the Multimedia Framework (providing framework support for cameras, still- and moving-image recording, replay and manipulation, and audio players) and the C Standard Library, an important support library for software porting.

The Base Services Layer

The foundational layer of Symbian OS, the Base Services layer provides the lowest level of user-side services. In particular, the Base Services layer includes the File Server and the User Library. The microkernel architecture of Symbian OS places them outside the kernel in user space. (This is in contrast to monolithic system architectures, such as both Linux and Microsoft Windows, in which file system services and User Library equivalents are provided as kernel services.)

Other important system frameworks provided by this layer include the ECom Plug-in Framework, which implements the standard management interface used by all Symbian OS framework plug-ins; Store, which provides the persistence model; the Central Repository, the DBMS framework; and the Cryptography Library.

The Base Services layer also includes the additional components which are needed to create a fully functioning base port without requiring any further high-level services: the Text Window Server and the Text Shell.

The Kernel Services and Hardware Interface Layer

The lowest layer of Symbian OS, the Kernel Services and Hardware Interface layer contains the operating system kernel itself, and the supporting components which abstract the interfaces to the underlying hardware, including logical and physical device drivers and 'variant support', which implements pre-packaged support for the standard, supported platforms (including the Emulator and reference hardware boards).

In releases up to Symbian OS v8, the kernel was the EKA1 (Kernel Architecture 1) kernel, the original Symbian OS kernel. In Symbian OS v8, the EKA2 (Kernel Architecture 2) real-time kernel shipped for the first time as an option. (It was designated Symbian OS v8.1b; Symbian OS v8.1a is

the Symbian OS v8.1 release with the original kernel architecture.) From Symbian OS v9, EKA1 no longer ships and all systems are based on the real-time EKA2 kernel.[9]

3.5 The Key Design Patterns

Probably the most pervasive architectural pattern in Symbian OS is the structuring client–server relationship between collaborating parts of the system. Clients wanting services request them from servers, which own and share all system resources between their clients.

Another widely used pattern is the use of asynchronous methods in client–server communications. Together, these two patterns impose their shape on the system. Like any good architecture, the patterns repeat at multiple levels of abstraction and in all corners of the system.

A third pervasive pattern is the use of a framework plug-in model to structure the internal relationships within complex parts of the system, to enable flexibility and extensibility. Flexibility in this context means run-time flexibility and is particularly important when resources are constrained. The ability to load the requested functionality on demand enables more efficient use of constrained resources (objects which are not used are not created and loaded). Extensibility is important too in a broader sense. The use of plug-ins enables the addition of behavior over a longer timescale without re-architecting or re-engineering the basic design. An example is the structure of the telephony system which encapsulates generic phone concepts which are then extended, for example for GSM- or CDMA-specific behaviors, by extension frameworks. The use of plug-ins also enables licensees to limit or extend functionality by removing or replacing plug-in implementations.

At a lower level, Symbian OS makes much use of specific, local idioms. For example, active objects are the design idiom which makes asynchronicity easy and are widely used. ('Asynchronicity' here means the ability to issue a service request without having to wait for the result before the thread of execution can continue.) Encapsulating asynchronicity into active objects is an elegant object-oriented design. (Active objects are examples of cooperative multitasking: multiple active objects execute in effect within the context of a single thread. Explicit multithreading is an example of non-cooperative multitasking, that allows pre-emption.)

Symbian OS has also evolved a number of implementation patterns, including 'leaving' functions and the cleanup stack, descriptors for safe strings, local class and member naming conventions and the use of manifest constants for some basic types.

[9] This history is described in detail in [Sales 2005], the in-depth, authoritative reference.

Symbian's microkernel design dates back to its original conception, but becomes even more significant in the context of the new real-time kernel architecture. The real-time architecture is essential for a system implementing a telephony stack, which depends on critical timing issues, and is also becoming increasingly important for fast, complex multimedia functionality. Together, phone and multimedia are arguably the most fundamental drivers for any contemporary operating system. As mobile phones, in particular, reach new levels of multimedia capability, to become fully functional converged multimedia devices (supporting streamed and broadcast images and sound, e.g. music streaming, two-way streaming for video phone conferencing and interactive broadcast TV), achieving true real-time performance has become an essential requirement for a phone operating system. The real-time kernel allows Symbian OS to meet that requirement, making it a suitable candidate for directly hosting a 3G telephony stack.

The real-time kernel architecture also introduces important changes (in particular to mechanisms such as interprocess communication) to support the new platform security model introduced from Symbian OS v9. (Strictly speaking, the security model is present in Symbian OS v8 but implements a null policy. The full security model, which depends on the new kernel architecture, is present from Symbian OS v9.)

The Client–Server Model

In Symbian OS, all system resources are managed by servers. The kernel itself is a server whose task is to manage the lowest level machine resources, CPU cycles and memory.

From the kernel up, this pattern is ubiquitous. For example, the display is a resource managed by the Window Server; display fonts and bitmaps are managed by the Font and Bitmap Server; the data communications hardware is managed by the Serial Server; the telephony stack and associated hardware by the Telephony Server; and so on all the way to the user-interface level, where the generic Uikon server (as specialized by the production GUI running on the final system) manages the GUI abstractions on behalf of application clients.

Threads and Processes

The client–server model interacts with the process and threading model in Symbian OS. While this is in keeping with a full object-oriented approach, which objectifies machine resources in order to make them the fundamental objects in the system, it can also cause confusion.

In Symbian OS, threads and processes are defined in [Sales 2005, Chapter 3] as follows:

- threads are the units of execution which the kernel scheduler schedules and runs
- processes are collections of at least one but possibly multiple threads which share the same memory address space (that is, an address mapping of virtual to physical memory).

Processes in other words are units of memory protection. In particular each process has its own heap, which is shared by all threads within the process. (Each thread has its own stack.)

A process is created as an instantiation of an executable image file (of type EXE in Symbian OS) and contains one thread. Creation of additional threads is under programmer control. Other executable code (for example, dynamically loaded code from a DLL file) is normally loaded into a dynamic-code segment attached to an existing process. Loading a DLL thus attaches dynamic code to the process context of the executing thread that invokes it.

Each server typically runs in its own process,[10] and its clients run in their own separate processes. Clients communicate with the server across the process boundary using the standard client–server conventions for interprocess communication (IPC).[11]

As Peter Jackson comments, Symbian OS falls somewhere between conventional operating system models in its thread and process model.

Peter Jackson:

Most of the threads versus processes issues are to do with overhead. In some operating systems, processes are fairly lightweight, so it's very easy to spawn another process to do some function and return data into a common pool somewhere. Where the process model is more heavyweight and the overhead of spawning another one is too great, then you invent threads and you let them inherit the rest of the process, so the thread is basically just a way of scheduling CPU activity. In Symbian OS, you can use whichever mechanism is appropriate to the requirements.

Server-Side and Client-Side Operations

Typically a server is built as an EXE executable that implements the server-side classes and a client-side DLL that implements the client-side interface to the server. When a client (either an application or another

[10] There are some exceptions for reasons of raw speed.

[11] [Sales 2005] defines the Symbian OS client–server model as inter-thread communication (ITC), which is strictly more accurate than referring to interprocess communication (IPC). However, arguably the significance of client–server communications is the crossing of the process boundary.

system service) requests the service, the client-side DLL is attached to the calling process and the server-side executable is loaded into a new dedicated process (if it is not already running).

Servers are thus protected from their clients, so that a misbehaving client cannot cause the server to fail. (The server and client memory spaces are quite separate.) A server has responsibility for cleaning up after a misbehaving client, to ensure that resource handles are not orphaned if the client fails.

At the heart of the client–server pattern therefore is the IPC mechanism and protocol, based on message passing, which allows the client in its process, running the client-side DLL, to communicate via a session with the server process. The base classes from which servers and their client-side interfaces are derived encapsulate the IPC mechanisms.

The general principles are as follows:[12]

- The client-side implementation, running in the client process, manages all the communications across the process boundary (in the typical case) with the server-side implementation running in the server process.
- The calling client connects to the client-side implementation and creates a session, implemented as a communications channel and protocol created by the kernel on behalf of the server and client.
- Client sessions are typically created by calling `Connect()` and are closed using `Close()` methods, in the client-side API. The client-side calls invoke the standard client–server protocol methods, for example `RSessionBase::CreateSession()` and `RProcess::Create()`. On a running server, this results in the client session being created; if the server is not already running, it causes the server to be started and the session to be created.
- The client typically invokes subsessions that encapsulate the detailed requests of the server-defined protocol. (In effect, each client–server message can be thought of as creating a subsession.)
- Typically, client-side implementations derive from `RSessionBase`, used to create sessions and send messages.
- Typically, the server side derives from `CServer`.

Servers are fundamental to the design of Symbian OS, and are (as the mantra has it) the essential mechanism for serializing access to shared resources, including physical hardware, so that they can be shared by multiple clients.

[12] The best description is [Stichbury 2005, Chapter 12].

Andrew Thoelke:

It's not so much that there is a server layer in the operating system as a hierarchy. It's very much a hierarchy and there are a lot of shared services. Some of them are shared by quite a few components and some of them really support just a very small part of the system, and of course those shared services may build on top of one or more client–server systems already.

Client–server is a deep pattern that is used as a structuring principle throughout the system.

Asynchronous Services

Another deep pattern in the system is the design of services to be asynchronous.

System responsiveness in a multitasking system (the impression that applications respond instantly and switch instantly) depends on asynchronous behavior; applications don't wait to finish processing one action before they are able to handle another.

The alternatives are blocking, or polling, or a combination of both. In a blocking request (the classic Unix pattern), the calling program makes a system call and waits for the call to return before continuing its processing. Polling executes a tight loop in which the caller checks to see if the event it wants is available and handles it when it is. (Polling is used by MS-DOS, for example, to fetch keystrokes from the keyboard.)

Blocking is unsatisfactory because it blocks others from accessing the system call which is being waited on, while it is waiting. Polling is unsatisfactory because code which is functionally idle, waiting for an event, is in reality not idle at all, but continuously executing its tight polling loop.

Blocking reduces responsiveness. Polling wastes clock cycles, which on a small system translates directly to power consumption and battery life.

Charles Davies:

Asynchronous services was driven by battery life. We were totally focused on that. For example on one of the Psion devices, we stopped the processor clock when it was idle. I don't know if that was innovative at the time. We certainly didn't copy it from anybody else, but we had a static processor. Usually in an idle process, the operating system is doing an idle loop. But we didn't do that, we stopped the clock on the processor and we turned the screen off, and that was fundamental to the design.

Typically, client–server interactions are asynchronous.

The Plug-in Framework Model

A final high-level design pattern, the plug-in framework model is used pervasively in Symbian OS, at all levels of the system from the UI Framework at the top to the lowest levels of hardware abstraction at the bottom.

A framework (as its name suggests) is an enclosing structure. A plug-in is an independent component that fits into the framework. The framework has no dependency on the plug-in, which implements an interface defined by the framework; the plug-in has a direct, but dynamic, dependency on the framework.

Frameworks are one of the earliest design patterns (going back to the time before design patterns were called design patterns, in fact) [Johnson 1998]. While, in principle, nothing limits them to object-oriented design, they lend themselves so naturally to object-oriented style that the two are strongly identified. A key principle of good design (again, not limited to object-oriented design but closely identified with it) is the separation of interface from implementation. On a small scale, this is what designing with classes achieves: a class abstracts an interface and its expected behavior and encapsulates its implementation. Frameworks provide a mechanism for this kind of abstraction and encapsulation at a higher level. As is often said, frameworks enable a complete design to be abstracted and reused.[13] Frameworks are therefore a profound and powerful way of constructing an object-oriented system.

In detail, a framework in Symbian OS defines an external interface to some part of the system (a complete and bounded logical or functional part) and an internal plug-in interface to which implementers of the framework functionality (the plug-ins) conform. In effect, the framework is a layer between a calling client and an implementation. In the extreme case, a 'thin' framework does little more than translate between the two interfaces and provide the mechanism for the framework to find and load its plug-ins. A 'thicker' framework may do much more, providing plug-in interfaces which are highly abstracted from the external visible client interface. Symbian OS contains frameworks at both extremes and most points in between.

Because in Symbian OS a framework exposes an external interface to a complete, logical piece of the system, most frameworks are also implemented as servers.

As well as providing interface abstraction and separation from implementation, and flexibility through decoupling, frameworks also provide a natural model for functional extension. This approach is used for example by the telephony-server framework to provide an open-ended design. The core framework supports generic telephony functionality based around a small number of generic concepts. Framework extensions implement

[13] A framework is 'reusable design' as [Johnson 1998] puts it.

the specialized behaviors which differentiate landline from mobile telephony, data from voice, circuit- from packet-switched, GSM from CDMA, and so on.

As well as this 'horizontal' extension of the range of functionality of the framework, such a plug-in also defines the interfaces which are implemented 'vertically' by further plug-ins that provide the actual services.

Because the plug-in framework model is pervasive, Symbian OS provides a plug-in interface framework. (Available since Symbian OS v7.0s but universally enforced since Symbian OS v8.0 as part of the phased introduction of Platform Security.) The plug-in framework (also known as ECom) standardizes the mechanisms and protocols that allow frameworks to locate and load the plug-ins which provide their implementations, and for plug-ins to register their presence and availability in the system as implementation modules.

Clearly, plug-ins pose a potential security threat because they provide a mechanism for untrusted (that is, externally supplied) code to be loaded into the processes of some system components (although the microkernel architecture keeps them well away from the kernel). The plug-in framework therefore enforces the security model on plug-ins before they are loaded [Heath 2006].

Another area in which plug-ins pose potential risks to the system is in performance. Potentially, a badly designed or poorly implemented plug-in can damage the performance of the framework that loads it. The plug-in model can also make it hard to understand the dynamic behavior of the operating system and, in particular, can make system-level debugging tricky, since the system can become (from the perspective of the debugger) highly indeterministic, unpredictable and unreproduceable.

However, enabling a pervasive model of run-time rather than static loading can boost system performance. Plug-ins are loaded on request; if they are not requested, they are not loaded, saving loading time and system resources (including RAM, on systems that do not provide execute-in-place).

An interesting example of just how pervasive the plug-in framework pattern is in Symbian OS is the original implementation of applications as plug-ins to the application and UI Framework rather than as more conventional executables. (This architecture changes somewhat in Symbian OS v9, where applications are implemented as EXEs rather than DLLs, while retaining other characteristics of plug-ins.)

In implementation terms, an ECom plug-in is implemented as a polymorphic DLL and a resource (RSC) file. The DLL entry point is a factory function that instantiates the plug-in object. All system plug-ins are stored into well-known locations, as required by the security model.

The plug-in framework provides a standard and universal mechanism for binding implementations (plug-ins) to interfaces (frameworks) at run

time, together with the mechanisms for packaging multiple interface implementations into a single DLL (that is, loading multiple implementations at once, to improve performance), plug-in registration and implementation versioning, discovery and loading including boot-time discovery optimizations to avoid run-time overhead, and cleanup after unloading plug-ins. (A plug-in instance cannot destroy itself, because its destructor code would be part of the code being removed from memory.) The framework also provides security-policy definition and policing mechanisms.

The plug-in framework is implemented as a server, in effect a broker between frameworks and conforming plug-ins, managing those plug-ins as a resource to its framework clients.

Microkernel Architecture

Symbian OS has a microkernel architecture, which sets it apart from operating systems such as Microsoft Windows and Linux.[14] In Symbian OS, core services that would be inside the kernel in a monolithic operating system are moved outside. The pervasive use of the client–server architecture, and the protection of system code from clients which follows from it, guarantees both the robustness and high availability of these services. The goal is a robust system that is also responsive and extensible; experience suggests that the design achieves it.

Andrew Thoelke:

The actual client–server architecture, the division into processes across the operating system and the boundary of the kernel, means that the actual privileged mode software is much smaller than in desktop operating systems. It's very nearly theoretical microkernel, but not completely truly microkernel because device drivers all run kernel side, and a true microkernel would say that device drivers should run user side, and who knows maybe we'll get there in a few years time. But all file system services, all higher level comms services including networking, and the windowing software for example, all run user side.

If anything the new EKA2 kernel architecture goes beyond the microkernel design and encapsulates the most fundamental kernel primitives within a true real-time nanokernel, supporting an extended kernel that implements the remaining Symbian OS kernel abstractions, but is equally

[14] There are microkernel implementations of Unix, based on the Mach microkernel. Mac OS X is an example; it is built as a Berkeley Unix variant with a Mach microkernel and proprietary user interface layer. Other microkernel designs include QNX, which is an operating system similar to Unix, but not Unix; Chorus, which is not just a microkernel but also object-oriented and which, like Mach, is capable of hosting Unix; and iTron, which is an important mobile-phone operating system in Japan.

capable of supporting 'personality' layers to mimic the interface of any other operating system. But the essential elegance of the Symbian OS kernel design goes right back to its earliest days.

Martin Tasker:

The Symbian model is that you're either a user thread or a kernel thread, and if you're a user thread then either you're an application thread, which has a session with the window server and interacts with the user, or you're a server thread which has no interaction with the user. And if you're a server thread, well then you sit around waiting for client requests to happen and when they do you service them, and in fact the kernel has a server and it does just that. There are a couple of kernel calls which are handled by something known as fast execs, which don't involve the kernel server. But the design philosophy of the kernel is to make those things very short and sweet and to put most of the work into the server. I think that's a cool architecture. Some of it goes down to Colly Myers's explainability requirement, that it takes more than an average programmer to implement any of this stuff, but any average programmer should be able to use it.

The lineage of course can be traced back to the precursor Psion systems.

Andrew Thoelke:

It owes its design very much to the heritage of Series 3. Colly Myers took that same OS structure, that you've got a small amount of protected mode software that can do everything, and that even all the file system and file services actually operate in a separate process from that and have less privileges, and that you have a very tightly integrated client–server architecture that actually binds everything together. That is definitely quite different to what you see in a lot of other systems.

Notwithstanding the move to the EKA2 kernel architecture, at a high level the lineage is still visibly present.

Martin Tasker:

The change from EKA1 to EKA2 is a hugely significant change. But at the system-design level, you know that change hasn't actually radically altered the system design at all. It's still either application processes or server processes, and that design was actually pioneered all the way back in SIBO, and it hasn't changed much since then, and the reason is: it's a proven design.

3.6 The Application Perspective

Symbian OS has been designed above all to be an application platform (although it might be argued that that has begun to change, and that in the latest devices it has become primarily an engine for driving fast, mobile data communications). Applications have always been an essential part of the system. The early operating system shipped with a complete set of productivity and communications applications targeting connected PDAs. Although Symbian OS no longer supplies a GUI and user-ready applications but only common application engines, Symbian OS phones now ship with more built-in applications than ever before, supplied either with the licensee GUI or as extras provided by the phone vendor or network operator.

Charles Davies:

Symbian started off as an operating system plus an application suite. We never designed it as an operating system independently of the suite of applications.

Just as importantly, both S60 and UIQ are also explicitly pitched as open platforms for third-party applications and provide extensive support for developers including freely available SDKs, support forums and tools.

From the beginning the approach to applications has been graphics-based. Like much else, the approach has evolved and, in particular, it has evolved as Symbian's user interface strategy has evolved. However, the principles of application structure have been essentially mature since the first release of S60 and UIQ in 2002.

Uikon is the topmost layer of Symbian OS. It provides the framework support on which a production user interface is built. The three currently available custom user interfaces are S60, UIQ and MOAP, but there is no engineering reason why any licensee should not build its own bespoke user interface, which indeed is precisely the origin of S60 and MOAP. Uikon abstracts application and control base classes in the Application Architecture and Control Environment class hierarchies to create generic GUI application classes (that is, classes free of a look and feel policy) which are customized by the custom user interface. The custom user interface abstracts the Uikon policy-free base classes to provide the policy-rich classes that applications derive from.

Uikon thus integrates the underlying support of the Application Architecture and the Control Environment to create a framework from which (as abstracted by the custom user interface), applications derive. Uikon is a framework and applications behave recognizably as plug-ins. Uikon is implemented as a server.

The Structure of an Application

Every application is built from three basic classes:[15]

- an application class, derived from the Application Architecture (`CApaApplication`)
- a document class, derived from the Application Architecture (`CEikDocument`)
- an application user interface class, derived from the Control Environment (`CCoeAppUiBase`).

These classes provide the fundamental application behavior. However, two important parts of the application are missing from this structure: the application view, which is the screen canvas the application uses to display its content, and the application data model and data interface implementations, which encapsulate the application 'engine'.

The classic application structure expects that the data model (the data-oriented application functionality) exists independently of the GUI implementation of the application and is, therefore, independent of any user interface classes. It is hooked into the user interface by a member pointer (`iModel`) in the document class. The classes specific to the user interface then interact with it purely through the APIs it exposes.[16]

Charles Davies:

We always had that structuring of applications, the idea of separating the UI from the application engine. That was an early design principle and it was the design guidance for application writers. We knew about Model–View–Controller, and we thought of an application engine as a model, and our design guidance was to keep the application logic separate from the UI. Not because we anticipated at that time multiple UI flavors, but because we recognized something more fundamental in terms of writing an application. That you might write an application and decide to improve the design of the UI, where the refinement of the UI was just pragmatic, the basic functional application logic stayed the same. So if you could separate those two things, that was good, and that led to the terminology of application engines.

[15] This is the 'classic' application structure, with roots in the Eikon applications of Psion Series 5. Both UIQ and S60 extend the design patterns for applications. See [Edwards 2004, p. 184] for discussions of the 'dialog-based' and 'view-switching' S60 application structure. UIQ applications also extend the basic pattern with custom view classes.

[16] This is in fact a very powerful design principle, implying, for example, that the data model can run without a direct user interface at all. Engines designed this way are independently testable and intrinsically highly portable between different user interfaces. The principle runs deep in the Symbian ethos, as witnessed by the presence of engines independent of the user interface in the operating system itself.

In Symbian OS, a control is a drawable screen region (in other words, the owner of screen real estate). The Application view class is derived directly from the Control Environment control base classes.

On small devices, where screen real estate is scarce, desktop-style windowing is not appropriate. A more natural approach for small displays is to switch whole-screen views, for example switching between a list-style view of contact names and a record-style view of the details of a single contact. Applications therefore typically define a hierarchy of views, with the main application view at the root.

Because Symbian OS is multitasking, multiple applications can be running at once, even though only one (the foreground application) will be presenting its view on the display. Both S60 and UIQ support switching directly between views in different applications, including launching the view of a new application inside the context of the current one (for example selecting a phone number from within a Contact entry and immediately switching to the phone application and dialing the number).

Symbian's application structure makes much of the detail of the application user interface programmable solely via resource files. Resource files are compiled separately as part of the application build process and linked into the built application, providing a natural mechanism for language localization (all text strings used within an application can be isolated in resource files and recompiled to a new language without having to recompile the application). Resource files are also compressed.

Charles Davies:

We lived in tougher times as far as Moore's law was concerned in those days. Resource files were around in contemporary GUI systems at that time. But from the beginning we did Huffman compression on resource files, and we were careful about the amount of information we put in them.

Uikon

The most striking fact about Symbian OS at the user interface level is its support for a replaceable user interface, and indeed the fact that it ships without a native user interface at all. (User-interface-dependent components are shipped only with a TechView test user interface.)

While it seems fair to say that Symbian did not get its user interface strategy right first time (in particular, the Device Family Reference Design (DFRD) strategy looks, with hindsight, to have been naïve), nonetheless the operating system has been able to support multiple licensees, each having a distinct user-interface philosophy, occupying different positions in the market and spanning diverse geographical locations. Those differences are encapsulated in the differences between the user interfaces that have evolved for Symbian OS.

S60 builds on the classic Nokia user interface to provide a simple, key-driven but graphically rich and arresting user interface. In contrast, UIQ is firmly pen-based and targets high-end phones with rich PDA-like functionality including pen-based handwriting recognition. MOAP aims squarely at its solely Japanese market, providing a graphically busy user interface featuring Kanji as well as Roman text and animated cartoon-style icons.

File System or 'Object Soup' Storage Model

FAT is the 'quick and dirty' file system that MS-DOS made famous. When work on EPOC started, the Apple Newton was a leading example of a different way to approach consumer computing (different, for example, from the MS-DOS-based Hewlett Packard machines which were the leading competitor for Psion's Series 3). Instead of a conventional file system the Newton employed an 'object soup' storage model.[17]

On any useful system, data requires a lifetime beyond that of the immediate context in which it is created, whether that means storing system settings, saving the memo you have just written to a file, or storing the contact details you have just updated.

Charles Davies:

We had a normal file system on the Series 3. When we went to C++, we talked a lot about persistent models of object-oriented programming, and we went for stream storage. We narrowly rejected SQL in favor of stream storage. I remember the design ideas around at the time, and it was done in the interests of efficiency. Different applications were having to save the same system objects and we were having to duplicate that code. So for something like page margins, which was a system structure, if that object knew how to serialize itself, that would solve the problem. You do that by having serialization within the object, so objects that might reasonably want to be persisted could persist themselves. And that was in the air, I mean Newton had its soup at that time which I think was object-oriented, and there was a belief at that time that object-oriented databases were it, and that objects ought to be seen as something that existed beyond the lifetimes of processes.

Objects, in other words, can be viewed as more than just the run-time realizations of object-oriented code constructs. However, in terms of the standards of the day, approaches based on something other than a file system were certainly the exception. The big challenge in maintaining data is that of data format and compatibility, ensuring that the data remains accessible. Any device which aims to be interoperable

[17] 'Object soup' is described in [Hildebrand 1994].

(in any sense) with other devices faces a similar challenge. In both cases, the design is immediately constrained in how far it can deviate from the data-format conventions of the day. For EPOC at that time, compatibility with desktop PCs was an essential requirement. For Symbian OS now, the requirement is more generalized to compatibility with other devices of all kinds. Probably the most important test case for both is readability of removable media file systems. (All other cases in which a Symbian OS device interoperates with another device can be managed by supporting communications protocols and standard data formats, which are independent of the underlying storage implementation.)

While external compatibility does not determine internal data formats, the need to support FAT on removable cards probably tipped the balance towards an internal FAT filing system. One (possibly apocryphal) story has it that the decision to go with FAT was a Monday morning *fait accompli* after Colly Myers had spent a weekend implementing it.

Peter Jackson:

There were periods when we explored all sorts of quite radical ideas but in the end we always came back to something fairly conservative, because if you take risks in more than one dimension at a time it doesn't work. So I spent quite a lot of time at one stage investigating an object-oriented filing system. But one day I think Colly Myers had a sudden realization and he just said, 'Let's do FAT', and he was probably right.

But FAT is not the whole story. In fact, Symbian OS layers a true object-oriented persistence model on top of the underlying FAT file system. As well as a POSIX-style interface to FAT, the operating system also provides an alternative streaming model.

It is an interesting fact that data formats, whether those of MS-Word or Lotus 1-2-3 or MS-Excel, have proved to be powerful weapons in the marketplace, in some cases almost more so than the applications which originated them. (The Lotus 1-2-3 data format lives on long after the demise of the program and, indeed, of the company.) Data in this sense is more important than the applications or even the operating systems with which it is created.

Peter Jackson:

The layout of the file is an example of a binary interface and, as software evolves, typically those layouts change, sometimes in quite an unstructured or unexpected way, because people don't think of them as being a binary interface that you have to protect. So the alternative way of looking at things is to say you don't think about that, you ignore the layout of the file. What you

do is you look at the APIs, and you program all your file manipulation stuff to use the same engines that originated the data in the first place.

In effect, this is the approach that Symbian adopted. But it has a cost.

Charles Davies:

We went for an architecture in which applications lost control of their persistent data formats, and in retrospect I think that was a mistake, because data lasts longer than applications. The persistence model is based on the in-memory aggregation in the heap of whatever data structure you're working with. For example, if it's a Contacts entry, then it consists of elements and you stream the elements. One problem is that if you try to debug it and you're looking at a file dump, its unfathomable. It's binary, it's compressed, so it's very efficient in the sense that when you invent a class it knows how to stream itself, so it's a sort of self-organizing persistence model, but the data dump is unfathomable. The second problem is that when you change your classes it changes how they serialize. So it works. But if you add a member function which needs to be persisted, then you change the data format. You lose data independence, and that stops complementers from working with your formats too. So we sacrificed data independence. And because that data has to carry forward for different versions of the operating system, you get stuck with that data format and you end up with a data migration problem. So I think that was a mistake. It would have been worth it to define data-independent formats. In my view that's what XML has proved, the XML movement has shown that data sticks longer than code.

In some ways, implementing a persistence model on top of a FAT system leads to the worst of both worlds, on the one hand missing out on the benefits of MS-DOS-style data independence, and on the other missing out on Newton-style simplicity.

Peter Jackson:

If you implement your permanent store structure in terms of a database design then you have all the advantages of being able to use database schema idioms to talk about what you're doing, and it turns out that those idioms now are fairly stable and universal. So I think there are examples where we have pruned away the databaseness of an application because we thought our customers didn't really want a database – but that may be a bad thing if one day our customers decide they want more than just flat data.

Store and DBMS

The native persistence model is provided by Store, which defines Stream and Store abstractions. Together they provide a simple and fully object-oriented mechanism for persistence:

- A Stream is an abstract interface that defines `Externalize()` and `Internalize()` methods for converting to and from internal and external data representations, including encrypted formats.
- A Store is an abstract interface that defines `Store()` and `Restore()` methods for persisting structured collections of streams, which represent whole documents. Store also defines a dictionary interface which allows streams to be located inside a store.

Symbian OS also includes DBMS, a generic relational database API layered on top of Store, as well as implementations including a lightweight, single-client version (for example, for use by a single application that wants a database-style data model which will not be shared with others). Databases are stored physically as files (single client databases may also be stored in streams).

Database queries are supported either through an SQL subset or a native API. Since the introduction of platform security, the DBMS implementation supports an access-policy mechanism to protect database contents.

3.7 Symbian OS Idioms

C++ is the native language of Symbian OS. Symbian's native APIs therefore are C++ APIs (although API bindings exist for other languages: OPL, Java and, most recently, Python). C++ is a complex, large and powerful language. The way C++ is used in Symbian OS is often criticized for being non-standard. For example, the Standard Template Library (STL) is not supported, the Standard Library implementation is incomplete, and POSIX semantics are only partly supported. Since Symbian OS competes with systems which do support standard C++, there is also little doubt that the operating system will evolve towards supporting more standard C++. But, like it or not, true native programming in C++ on Symbian OS requires understanding and using its native C++ idioms.

Among some developers inside the company the view has been unashamedly one of, 'Those who can, will; those who can't should use Java, Python, or even OPL'.[18] While that may not make for mass market appeal for Symbian C++ itself, the fact is that programming on

[18] For example, see the remarks by David Wood in Chapter 18.

any platform requires specialist expertise as well as general expertise, and, in that, Symbian OS is no different. The skill level required is commensurate with the programming problem. It is far from easy to write software for consumer devices on which software failures, glitches, freezes and crashes – things people put up with regularly on their PCs – are simply not an option. Mobility, footprint, battery power, the different user expectations, screen size, key size and all the other specifics of their small form factors make mobile devices not at all like desktop ones; phones, cameras, music players and other consumer devices are different.

Symbian OS idioms are not casual idiosyncrasies; they are deliberate constraints on the C++ language devised to constrain developer choices, consequences of the market the operating system targets, and of the embedded-systems nature of ROM-based devices. Strictly speaking, they are less architectural than implementational but, in terms of the overall design, they are important and they have an important place in the history of the evolution of the system. Understanding them is essential to understanding what is different about Symbian OS, and what is different about mobile devices. There are some large-scale differences.

- Lack of a native user interface means that the development experience is significantly different for device creation developers using the TechView test user interface than for developers later in the product lifecycle using S60, UIQ or MOAP.
- The build system is designed for embedded-style cross-compilation, which is a different experience from desktop development.
- Idioms have evolved to support the use of re-entrant, ROM-based DLLs, for example disallowing global static data.
- Other optimizations for memory-constrained, ROM-based systems result in some specific DLL idioms (link by ordinal not name, for example).

There are what might be described as language-motivated idioms:

- descriptors
- leaving functions
- the cleanup stack
- two-phase construction.

And there are some design-choice idioms:

- active objects and the process and threading model
- UIDs

- static libraries and object-oriented encapsulation
- resource files to isolate locale-specific data, for example, text strings.

Active Objects

Active objects are an abstraction of asynchronous requests and are designed to provide a transparent and simple multitasking model.

An active object is an event handler which implements the abstract interface defined by the `CActive` class and consists of request and cancellation methods, which request (or cancel) the service the object should handle, and a `Run()` method which implements the actual event handling. When the requested service completes and there is a result to be handled, a local active scheduler invokes the active object's `Run()` method to handle the completed event.

An active scheduler is created by the UI Framework for each application. All active objects invoked by an application (but only that application's active objects) share a single thread, in which they are not pre-empted (i.e. they are scheduled in priority order by the scheduler).

Active objects are a pervasive Symbian idiom and provide a non-pre-emptive multitasking alternative to explicitly creating multithreaded programs (although that option remains available to developers), as a solution to the problem of managing multiple paths of execution within a program, in the context of an event-based, reactive application model. From the perspective of a GUI application developer they offer a much easier solution than multithreading, in effect handing off the awkward details to the system.

Charles Davies:

Our model for events was very much asynchronous events and signals and requests. So what we had first of all, and it's what other systems have too, is that you make one or more requests for events, and events include timers and serial events and all kind of events that can come out of anywhere, not just user-originated events. So you just set off a large number of events and then you wait for any one of them to come through. So things need to be able to respond to events from multiple sources. Now Windows had a way of handling this. There's a Windows API, though it's not very elegant. The problem is, it's tied to the GUI programming model. In Windows you have to run up the whole GUI to get the event model going, and we thought that was a real weakness in mobile devices. We thought that servers needed this as well, that servers sit there waiting for events from multiple sources, events like 'my client has died', which comes from a different source than the message channel saying 'here's the next request from the client'.

The event-driven model is essentially a state-machine model. But, except within niche areas such as communications programming, these

were not widely used patterns, especially for applications programming. And except for those familiar with Windows at the time, or with other GUI systems such as Amiga and Macintosh, the event-driven application model was not widely or well understood.

Charles Davies:

When I was interviewing people I used an example of a terminal emulation program. Here is a program that indisputably gets events not just from the user. The normal, naïve way of writing an interactive application at that time would be to wait for a keypress, see what keypress it was, and respond to it; was it a function key, was it any other key? You'd have some horrible case statement responding to a keypress. So I would ask, 'How would you write an application where you don't know whether your next input is coming through the serial port or from the keypress?' And if they had a good answer to it they got hired, and if they didn't, they didn't.

Well we started off programming it the way that anybody would program it, you make asynchronous requests on whatever event sources you want to respond to. There are many pitfalls in doing that, for example if you don't consume that event in the right way. You end up with an event loop that's quite messy, and it's pages long, and people were making mistakes. Every event loop was buggy, and horrible bugs too, so we said 'Let's make it modular.'

Martin Tasker had the benefit of a background of programming IBM mainframes:

Martin Tasker:

I've written plenty of event-handling loops, in communications programs or command handlers where by definition you don't know what's going to happen next. Every time I wrote one of these loops I remember thinking, 'Have I got this right?' Dry running through every possibility, you used to have to tell people coming on to the team, 'No, if you handle your loop that way you're either going to double-handle some event or fail to handle some event, or you're not going to handle event number 2 if event number 2 happens while you're handling event number 1, or you're not actually going to handle event number 2 until event number 3 comes along. . .' These are all mistakes that everybody makes when they're writing event-handling programs. Over the lifetime of a program you tend to add in more and more events, or you remove them, and you change things around. And in those circumstances, when you're modifying existing code, it's tremendously difficult to get event-handling loops right.

Active objects were devised explicitly to solve such problems, by creating an easy-to-understand and easy-to-use mechanism for firing

off event handlers asynchronously, deliberately breaking the dependencies between events which are implied by the big, single-block switch statement which is the typical implementation. More generically, active objects enable multitasking within applications without the use of explicit multithreading.

Charles Davies:

We could have done it with threads and created a multithreaded UI, which by the way is what Java does. But the bad thing about threads is that you can pre-empt at any time, and then you've got to protect the data, because you have no idea when you're processing one thread what state the data is in. The solution was active objects, for any program that responded to events from multiple sources. So it came about because people were getting it wrong, because the old way was so complicated. So what are active objects? They're really non-pre-emptive multitasking within an application. And that is a very strong pattern. But it is also something that throws people, because it wasn't copied. It was invented here, and it's widely used, and it has been useful, but it is a particular strength of Symbian OS.

Active objects are used widely throughout the operating system, as well as providing a ready-made mechanism for developers creating native Symbian OS applications.

Martin Tasker:

Colly Myers was right, active objects are a fantastic solution. For people who know they are dealing with event-handling programs, they are an absolute joy. And the whole single-threaded nature of an application process is also great for programmers. In an event-handling system, active objects are a natural way of handling things, and they are easier for programmers to work with than pretty much all of the alternatives.

Cleanup, Leaving and Two-Phase Construction

The native Symbian OS error-recovery model evolved explicitly to handle the kinds of errors that should be expected on resource-constrained and mobile devices: low-memory situations, low-power situations, sudden loss of power, loss of connectivity or intermittent connectivity, and even the sudden loss of a file system, for example when a removable media card is physically removed from the device without unmounting. These are all likely or even daily occurrences in the mobile phone context, causing errors from which the system must recover gracefully. In contrast, for a large system these may be rare enough occurrences for system failure with an 'unrecoverable error' message to be acceptable.

The Symbian OS model is proven, playing a large part in the unrivaled robustness of the system, and going back to the earliest days of the operating system, and indeed to Psion systems before it.

Charles Davies:

We had `Enter()` and `Leave()` in the 16-bit system, which was Kernighan and Ritchie inspired. When we went to C++, the standards for exception handling were still being written, so they certainly weren't available in compilers. So we carried forward `Leave()` and `Enter()` rather than adopting native C++ exception handling, because at that time it consisted of `longjump()` and `setjump()`. It was very unstructured, and we didn't like that. We liked `Enter()` and `Leave()`, and we stuck with it.

In Symbian OS, `Leave()` is a system function (provided by the User Library) which provides error propagation within a program. Typically, `Leave()` is used to guard any calls which can fail (for conditions such as out of memory, no network coverage and disk full). The system unwinds the call stack until it finds a prior `Leave()` call wrapped by a TRAP macro, at which point the TRAP is executed and the failure is handled by the program in which it occurred.[19]

Functions which may fail because of a leave, whether because they directly invoke the action which might fail or do so indirectly by calling some other function that does, are described as 'leaving' functions. By convention, leaving functions are named with a trailing 'L', which makes it easy for programmers to see where they are invoked and trap appropriately.

The second leg of the error-handling strategy uses the 'cleanup stack' to store pointers to heap-allocated objects whose destructors will fail to be called if the normal path of program execution is derailed by a leave.[20] As well as unwinding the call stack to handle the leave, the cleanup stack is also unwound and destructors are called on any pushed objects.

The third leg of the strategy is 'two-phase construction', which guarantees that C++ construction of an object will always succeed, by moving any leaving calls out of the C++ constructor into a secondary constructor. (It is important that construction succeeds, since only then can the object's destructor be called; if the destructor cannot be called, memory may have been leaked [Stroustrup 1993, p. 311].) Again, a number of system functions are available to regularize the pattern and take care of underlying details for developers. (In its earliest implementation, two-phase construction was matched by two-phase destruction. The eventual consensus was that this was an idiom too far.)

[19] See [Stichbury 2005, p. 14] for a detailed explanation.

[20] See the discussion in [Harrison 2003, p. 150]. This is the authoritative programmers' guide.

Charles Davies:

We had an ethic that said that memory leakage was something the programmer was expected to manage. So something like the Window Server, which might be running for a year at a time, needed to make sure that if an exception was called it didn't leak memory. The cleanup stack was an invention to make it easier for people to do that. You'd have an event loop, and at the high end of the event loop you'd push things on the stack that needed to be unwound, whether they were files that needed to be closed or objects that needed to be destroyed. That was a pragmatic thing, you know. 'Let's provide something that encourages well-written applications from the point of view of memory leakage.'

Cleanup is pervasive in the system ([Harrison 2003, p. 135]), permeating every line of code a developer writes, or reads, in Symbian OS, with its highly visible trailing 'L' naming convention, its `Leave()` methods and TRAPs, and its cleanup stack push and pop calls.

For new developers, it is both highly visible and immediately unfamiliar, which leads to an immediate impression that the code is both strange and difficult. However, the conventions are not intrinsically difficult, even if the discipline may be. The purpose is equally straightforward: to manage run-time resource failures. On a small device, memory may rapidly get filled up by the user (whether by loading a massive image, downloading too many MP3s, or simply taking more pictures or video clips than the device has room for). Other resources, whether USB cable connections, infrared links, phone network signals, or removable media cards, can simply disappear without warning at any time. Mostly these hazards simply do not exist on desktop systems. On phones, they are the norm.

Martin Tasker:

I think the cleanup stack was a brilliant solution to the problem that we were faced with at the time.

Descriptors

Descriptors are the Symbian OS idiom for safe strings. ('Safe' means both type safe and memory safe and compares with C++ native C-style strings, which are neither[21]) Descriptors were invented (by Colly Myers) because there was no suitable C++ library class, or none that was readily available.

[21] Nor are Java or Microsoft Foundation Class strings for that matter, according to [Stichbury 2005, p. 55].

In principle, descriptors simply wrap character-style data and include length encoding and overrun checking. (Descriptors are not terminated by NULL; they encode their length in bytes into their header, and refuse to overrun their length.) As well as this basic behavior they also provide supporting methods for searching, matching, comparison and sorting.

Descriptors support two 'widths', that is, 8-bit or 16-bit characters, based on C++ `#define` (`typedef`) and originally designed to enable a complete system build to be switched, more or less with a single definition, between ASCII-based and Unicode-based character text support.

More interestingly, descriptors also support modifiable and unmodifiable variants and stack- and heap-based variants. The content of unmodifiable (constant) descriptors cannot be altered, although it can be replaced, whereas that of modifiable descriptors can be altered, up to the size with which the descriptor was constructed.[22]

Another important distinction is between buffer and pointer descriptor classes. Buffer descriptors actually contain data, whereas pointer descriptors point to data stored elsewhere (typically either in a buffer or a literal). A pointer descriptor, in other words, does not contain its own data. A final distinction is between stack-based and heap-based buffer descriptors. Stack-based descriptors are relatively transient and should be used for small strings because they are created directly on the stack (a typical use is to create a file name, for example. Heap-based descriptors, on the other hand, are intended to have longer duration and are likely to be shared through the run-time life of a program (see Table 3.1).[23]

Table 3.1 Descriptor classes.

	Constant	**Modifiable**
Pointer	`TPtrC`	`TPtr`
Buffer (stack-based)	`TBufC`	`TBuf`
Heap-based	`HBufC`	

See [Harrison 2003, p. 123] for a fuller explanation of the descriptor classes.

[22] Although modifiable, once allocated there is no further memory allocation for a descriptor, so its physical length cannot be extended. For example, to append new content to a descriptor requires that there is already room within the descriptor for the data to be appended.

[23] [Stitchbury 2005] contains a good overview.

Descriptors differ from simple literals, which are defined as constants using the_LIT macro, in that they are dynamic (literals are created at compile time, descriptors are not). A typical use of a pointer descriptor is to point to a literal.

Martin Tasker:

The 8-bit/16-bit aspect was ASCII versus Unicode, though, in retrospect we should have been braver about adopting Unicode straight away. But bear in mind that the ARM 3 instruction set we were then using didn't have any 16-bit instructions or, more accurately, it didn't have any instructions to manipulate 16-bit data types, so it was not efficient to use Unicode at that time. But maybe we should have had more foresight and courage, because it turned out to be a distraction. But as a kind of memory buffer, I think they were reasonably distinctive.

Given the state of the art at the time, Peter Jackson believes that the distinction between 8-bit and 16-bit was understandable but that a more naturally object-oriented approach would have been preferable.

Peter Jackson:

I think it would have been more elegant to have a descriptor that knew internally what kind of descriptor it was, whether it was the 8-bit or 16-bit variant. I never liked the fact that some of these things were done by macros.

Descriptors are not only type safe, they are memory safe, making memory overflow ('out-of-bounds' behavior) impossible. Descriptor methods will panic if an out-of-bounds attempt is detected (see Figure 3.1).

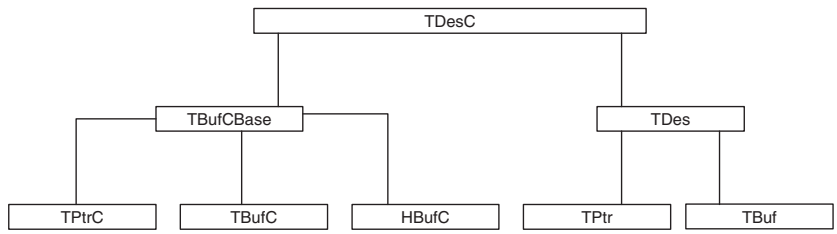

Figure 3.1 Descriptor class hierarchy

Charles Davies:

Descriptors were Colly Myers's thing, definitely, and the idea was rather like the cleanup stack, to stop people doing memory overwrites. That's a big protection against worms and other attacks, deliberate and malicious overwriting of the heap, although at the time that wasn't the driving reason to do it. We did it to stop programmers making mistakes.

C and T and Other Classes

As well as the use of the trailing 'L' (for 'leaving') and 'C' (for 'constant') to flag properties of methods, Symbian OS also uses some similarly straightforward class-naming conventions to flag fundamental properties of classes.

Martin Tasker:

If you look at the C and T types, they offer a very, very simple guide to the programmer as to how to use these types. They are as simple as Java's objects and built-ins. We don't do garbage collection because C++ doesn't do garbage collection, so we have to cope with that. We have to do it manually, but otherwise I think our conventions are as simple as Java.

The most important naming conventions are summarized as follows: [24]

- T classes are simple types which require no destructor and behave like C++ built-in types.
- C classes derive from `CBase` and should always be explicitly constructed, thus ensuring that they are always allocated on the heap. `CBase` classes also therefore require explicit destruction. `CBase` provides a basic level of additional support, including a virtual destructor, allowing `CBase`-derived objects to be deleted through the `CBase` pointer and performing cleanup stack housekeeping. `CBase` also overloads operator `new` to zero-initialize an object when it is first allocated on the heap. All member data of derived classes is therefore guaranteed to be zero on initialization.
- R classes indicate resource classes, typically a client session handle for a server session. Since an R class typically contains only a handle, it does not require either construction or destruction. R classes therefore may safely be either automatics or class members.

[24] [Stichbury 2005, Chapter 1] provides a comprehensive discussion.

- M classes are 'mixin' classes (abstract interface classes), the only form in which multiple inheritance is supported in Symbian OS.
- Descriptors are immediately recognizable as either `TPtr` pointer descriptors, or `TBuf` (stack-based) or `HBufC` (heap-based) buffer descriptors.

Manifest Constants

Symbian OS uses manifest constants – implemented as `typedef`s, that is, system-defined types – instead of the native types supported by a standard C++ compiler on standard hardware. This is partly, of course, because the cross-development model means that the eventual intended target platform is not the same as the development platform, hence the 'native' types of the platform on which the code is compiled may differ from those of the platform on which it is intended to run. The use of type definitions also has its roots in designing to support both ASCII and Unicode builds, which is now superfluous since Symbian OS has been all-Unicode since before v6.

Supporting emulator builds (that is, running Symbian OS programs on PC as well as ARM, and not just developing on PC) creates the additional complexity of requiring not one supported compiler but two (or more); originally Microsoft compilers were specified for emulator builds and GCC for ARM. More recently Metrowerks and Borland compilers have been supported and, in Symbian OS v9, ARM's RVCT replaces GCC as the 'official' ARM target compiler (although GCCE is still supported to ensure a low-cost development option). Recent initiatives such as Eclipse, for example, or the adoption of the standard ARM EABI are likely to continue to change the story of the development tools.[25] Again, using manifest constants provides the necessary level of decoupling of code from compiler dependencies.

The key classes are summarized as follows:[26]

- `TInt` and `TUint` are the generic types for signed/unsigned integer values; `TInt8`, `TInt16`, `TInt32`, and `TUint8`, `TUint16`, `TUint32` are also provided; in general, the least specific types are preferred, that is, `TInt` and `TUint`
- `TInt64` is a 64-bit integer type intended for platforms without a native 64-bit type

[25] Symbian, like Psion before it, has always assumed that mainstream development is done under Microsoft Windows, although this is not the only solution that works. There are a number of independent open-source solutions for developers wanting to work on Linux or Mac OS X.

[26] Again, [Stichbury 2005, Chapter 1] provides a comprehensive discussion.

- `TReal`, `TReal32` and `TReal64` are single- and double-precision floating-point types; again the least specific type, `TReal`, is preferred
- `TText8` and `TText16` are 8-bit and 16-bit unsigned types for characters
- `TBool` is a 32-bit unsigned Boolean type
- `TAny*` is used instead of void*.

Unique Identifiers

Unique identifiers (UIDs, implemented as signed 32-bit values) are centrally controlled in Symbian OS. One common usage of them is to identify applications and other binary and data types. UIDs, for example, are used in Symbian OS to associate data types with programs and plug-in types with frameworks. UIDs are also used as feature IDs and package IDs (for SIS files).

> **Charles Davies:**
>
> The idea was that if you had polymorphic DLLs, dynamic libraries in other words, then there are situations where the DLL is a plug-in, and it all goes very wrong if the caller doesn't get the interface it's expecting from the DLL, so we needed to characterize the interface. And we came up with the idea of using a UID to do that.

UIDs are used in a three-tier construction to build `TUidType` objects:

- UID1 – a system level identifier that distinguishes EXE from DLL types
- UID2 – a specifier for library types that distinguishes between shared library DLLs and various types of polymorphic DLL (for example FEPs and other types of plug-in)
- UID3 – the individual component ID, also used by default as the secure identifier (SID) required by platform security.[27]

UID3 is used, for example, by developers to uniquely identify their applications, and can then be used by the streams, stores and files created by that application to identify themselves. UID3 is assigned through Symbian's UID allocation database, from which third-party developers can request blocks of UIDs for use in their applications.

Platform Security introduces two new types of UID, the SID (Secure ID), which by default is identical to UID3, and VID (Vendor ID).

[27] See the discussion in [Sales 2005, p. 328].

3.8 Platform Security from Symbian OS v9

Platform Security is the system-wide security model introduced in Symbian OS v9. Providing an open, third-party programmable platform has been an important principle in the development of Symbian OS. However, openness brings with it the risk of misbehaving software (whether accidentally or deliberately misbehaving) finding its way onto users' devices. The security model is designed to protect users from that risk, while still preserving the openness of the platform.

Architecturally, Platform Security is a set of pervasive changes at all levels of the system, based on a simple conceptual model,[28] which is deliberately as lightweight as possible, and supported by the Symbian Signed certificate signing program, which provides a means for creating a formal link between an application and its origin, as well as providing a review mechanism to promote best practice in designing and writing Symbian OS applications.

Will Palmer is one of the system architects who is currently responsible for the Platform Security project.

Will Palmer:

There are three principles to Platform Security. The first principle is the unit of trust, the idea of the process being the unit of trust. Since memory is already protected per-process on the processor, that fits quite nicely, and it also has the advantage of being a 'least-privilege' approach, based on the smallest element in the operating system. The second principle is the idea of capabilities, which are in effect authorization tokens. So to be able to access a potential resource, a process needs to possess a particular capability that allows it to do so. And the third principle is data caging, which is about read and write protection of files, which protects the integrity of data as well as protecting data from prying eyes.

The essential principles are:

- processes as the unit of trust,[29] which turns trust into another process-granular system resource
- capabilities as the tokens of trust, which are required to perform actions

[28] According to [Heath 2006, p. 18], the model conforms to the eight design principles of [Saltzer and Schroeder 1975], which include economy, openness, least privilege and psychological acceptability.

[29] This is an elegant extension of the kernel's process model, in which the process is the unit of ownership of all system resources (for example, memory protection is per process).

- data caging, which protects data from prying eyes (by policing read access) or interference (by policing write access) or both.

The direct consequence of defining the process as the unit of trust is that all threads in a process share the same level of trust (which is natural, since they have access to the same resources).

The goal is to protect device users from the kinds of intentionally rogue software, or 'malware', that plague the PC world. Symbian OS for a long time avoided some of the worst threats from malware because it was typically deployed in ROM-based devices, in which the system itself cannot be corrupted (for example, it is impossible to install trapdoors or trojans in system files) because system code is stored in unwriteable ROM memory. By design, Symbian OS also protected against some of the more trivial security holes found on other systems. Descriptors, for example, make buffer overrun attacks much harder. Similarly, Symbian's microkernel architecture helps to increase security and robustness; since the trusted kernel is deliberately the smallest possible subset of system functions, there is little privileged code to exploit, and the smaller codebase is easier to review and validate.

The nature of mobile devices, especially phones, also makes them different from desktop systems. The physical access model is different (personal devices are less likely to be shared) and the network access models are different (connections are transient).

On the other hand, phones also present new opportunities for malware. If a phone, or user, can be spoofed into making a call, real money is at stake. (Premium-rate-phone-number scams are an example.) From a network perspective, the cost of network disruption is immediately commercially quantifiable in a way that Internet attacks are not.

These differences all require appropriately designed security mechanisms.

Will Palmer:

When the capability model was designed there were a set of constraints about what it had to deliver: it had to be robust; it had to be simple; and it shouldn't get in the way of the operation of a phone so, for example, you couldn't use hundreds of extra clock cycles on it, because on a small device you have performance and power constraints. Also it had to be appropriate for an open operating system: people have to be able to install additional software on their phones and it has to be simple and easy to understand.

Data caging, for example, was chosen for its simplicity and economy (in terms of clock cycles and power). Another important consideration was that mechanisms which users are quite comfortable with on desktop computers – logging on, for example – would be quite inappropriate on a phone.

Will Palmer:

Authorization based on the process–capability model is simple to understand and it fits the phone case much better than an authentication system. So in an authentication system you log on and your password authenticates you to the system, and once authenticated you can do anything permitted by your authentication level. But a phone is different: it's a single-user environment; it's in your pocket; it belongs to you. Although things are getting more complex now because of requirements coming in for administrative rights. For example, the network operator might want to change settings on the phone.

The capability mechanism is used to protect both 'system' and 'user' (i.e., application-owned) resources. Will Palmer sums up the difference neatly.

Will Palmer:

It's not that some types of capabilities are more powerful than others, they just protect different things. System capabilities protect the integrity of stakeholders and of the device, whereas user capabilities protect the user's privacy and money.

Protected APIs are tagged at method-level with the capability required to exercise them and access any underlying resources (data files, for example). The capabilities of a method are part of its interface. To use protected APIs therefore, developers must request an appropriate set of capabilities, which is done through the Symbian Signed program.

A 'signed' application is granted a set of capabilities. Application capabilities are verified by servers when protected APIs are called by applications. Unsigned software is flagged to the user at installation time as being unsigned (and therefore untrusted). Thus, while unsigned applications can assign any user capabilities to any binaries as they see fit, the user is alerted at installation time and given the option to approve the application or not. Unsigned applications cannot use system capabilities, in other words they cannot use APIs which affect the behavior of the device. Data security is provided on a per-application basis by the data-caging model.