14

The Use of Object-oriented Design in Symbian OS

14.1 Introduction

Symbian OS traces its lineage back to the operating systems that Psion created for a succession of innovative and market-leading handheld devices, from the early Organisers through to the Psion Series 3 and, finally, to the Psion Series 5, for which the first versions of what became Symbian OS were designed. Psion was an early adopter of object-oriented programming techniques and an early adopter of C++, opting to build a commercial, production operating system in C++ significantly ahead of the mainstream.

This case study explores that history, and the consequences. It goes on to survey some of the ways in which object-oriented techniques are used in Symbian OS, for example the adoption of model–view–controller (MVC) as the basis for the application model, the widespread use of frameworks, the active object idiom, and the way object-oriented ideas influenced the design of the kernel itself.

The adoption of object orientation, generally, and of C++, in particular, were radical steps for the company; both posed challenges, both were motivated by some clear expected benefits. Within the company at the time, both decisions were controversial and the risks were enormous. The history and the controversies are instructive and still relevant today – both within and outside Symbian. They give a flavor of the company's particular character at that time, of its openness, its willingness to take risks, its commitment to understanding what the 'right thing' was to do, and then doing it; as well as giving a flavor of what the broader context was for the design of Symbian OS and of some of the influences that shaped it.

14.2 Pioneering the Object Approach in Psion

The operating system that eventually became Symbian OS started life as Psion's 32-bit, 'next generation' operating system, an all-new operating system designed for a new generation of 32-bit hardware which was planned to replace the then current (and highly successful) Series 3 range of 16-bit palmtops. Psion in early 1994 already had a history of experimenting with an object-based design approach and object-oriented programming techniques. The 16-bit operating system for the Series 3 (SIBO), for example, had used object-oriented design ideas and implementation techniques heavily at the application and user-interface levels, with great success. The Series 3 had a powerful but easy-to-use graphical user interface (GUI), unique for a machine of its class. Its user-interface and application architectures were fully event driven, with an object-oriented implementation. Martin Tasker, new to the company, saw object orientation as a perfect match for the problem.

Martin Tasker:

They had understood that objects helped you to do GUIs and systems, and in the SIBO GUI they had the perfect event-driven system.

Object-Flavored C

The big question facing the company as it prepared to start work on the new 32-bit system was how best to build on its object-oriented legacy. An important focus of the debate was which object-oriented language to choose.

Peter Jackson:

The big debate that was going on was, 'Do we write another proprietary object system like we did for SIBO or do we go to C++?' People were saying, 'C++ is the next big thing, everyone will know how to do C++ programming, therefore it's easier to get engineers without special training.' And of course this is complete nonsense, because what you really need to know is not C++, but how to program against a particular object model and the APIs that have been developed for the system you're programming. It's just as big a learning curve as learning some proprietary object framework. But the C++ faction won out.

For the 16-bit system, Psion had evolved a proprietary object-flavored dialect of C, along with tools which pre-processed this 'C with classes' code and generated standard C output, which was then passed into

a conventional compile and link stage. Objective-C was the explicit influence.[1]

In fact, this was the second time around for the C++ debate. At the outset of the 16-bit project, the team had evaluated a number of language options including C++, before deciding to develop its own object flavored, C-based solution.

David Wood:

We looked at C++ as one alternative, but we were concerned by a number of things. First, we saw it as being a very large language compared with C and we thought that we would just have no constraints on our design. Second, we saw it as still being immature. There weren't mature compilers for it at that stage, certainly not for the PC which was our development platform. So we took the view that it would be a big, big jump to adopt C++ and we didn't know much about it, whereas we thought we could do a more constrained job ourselves.

Psion's solution was home-grown and decidedly arcane, but on the other hand it did what it was meant to do and it did what was needed. Since the users were all in-house, the fact that it was a proprietary solution was not an issue. It also gave the team complete control of the most important components of the development toolchain. There was, however, rather more to the detail than simply pre-processing pseudo-class definitions into standard C.

Andrew Thoelke:

You had a class description file which basically said, 'This is a class' and it knew what its base class was, it knew what methods you had added, what methods you had overridden, and it knew its data members. That was then processed by a preprocessor, in effect a class compiler, which generated a C header file with lots of `#defines` in it which defined method numbers and offsets of data and that sort of thing. It also defined data structures for the classes themselves, so you could actually use pointers to the class objects, and then it generated an object file which contained a binary representation of the class structure. These were then all compiled together, so in a given dynamic library in SIBO the header contained initially a lot of information describing all the classes within that dynamic library, and then following that was all the

[1] Objective-C (see Chapter 4) is an object-oriented superset of C with a Smalltalk-style (infix) message syntax, which emerged in the early 1980s more or less concurrently with the Cfront versions of early C++, in an independent effort to harness the plain syntax and underlying efficiency of C to an object model. Objective-C is still in use as the native system programming language for Mac OS X, having been inherited by it from the NextStep operating system.

actual code itself, the actual functions, the methods. And normally you would have written that in plain C.

High-level object-oriented design could therefore be directly implemented using class-like constructs. Underneath, the class (or pseudo-class) method implementations were vanilla C. The class compiler was known officially as the Category Translator.

David Wood:

It looked at the class definition file and generated C output. There were also helper functions in the operating system which enabled indirect function calling to methods in the classes you defined, which was the equivalent of C++ virtual tables. Our implementation didn't use virtual tables in the same way as C++. We tried to implement that in a more efficient way, driven by the desire to save memory.

The importance of saving memory in order to maximize the small ROM and RAM footprint of the hardware was deeply engrained in the Psion culture. Necessarily so, since early Series 3 models were based on either 128 KB or 256 KB RAM with 512 KB ROM, increasing in later models to 1 MB or 2 MB RAM with1 MB ROM, and to 2 MB ROM for the last model in the series.

Another optimization was the replacing of the standard C library with a dedicated library of support functions implemented as thin wrappers around native operating system calls. This kept the dynamic libraries small because, instead of each library including the same several kilobytes of standard library code, they simply made direct calls to operating-system functions. It also improved run-time performance, another important driver.

Peter Jackson:

All in all, it made for some very lean and mean code and it was also a very elegant system. It was precisely targeted to meet the requirements of the object space you would need to have for a PDA.

Throughout the system, assembly code was also extensively used to achieve speed and small code size. Again, this code was written inside an object framework which integrated with the C-based object model.

Martin Budden:

The assembler stuff was written to the same kind of interface, so it was effectively object-oriented assembler, and it fitted in the same framework, at least at the application level.

While object-oriented techniques were used extensively in the higher levels of the system, the lower levels of the system were more conventional. The kernel implementation, for example, and the lower-level system services, such as the file server, were written in conventional C.

Geert Bollen:

It was a layered OO [object-oriented] mechanism, using message dispatch on top of a C system. That was the Psion tradition. Underneath was a classic C-API-based operating system with a small number of system calls exposed as C functions.

This is exactly how it should have been, in David Wood's view. Object orientation was a pragmatic choice, not an ideological one. It was intended to achieve some clearly defined benefits without undue risk either to the system (in terms of size and performance, for example: code bloat was the big fear) or time-to-market (which a wholesale jump to C++ might have jeopardized).

David Wood:

OO came in stages, which is how all large software systems should evolve. One of the very important rules of software, or indeed of anything else in life, is, If you're not sure what you're doing, don't do it on a large scale. I think the phrase is attributable to Tom Gilb. What it means is that you should go and experiment, and you should evolve through iterations. So we introduced objects first of all just for the user interface, and then we realized that we could apply those ideas elsewhere too. So we proceeded through iterations.

C++ Language Choice and Adoption

At the outset of the 32-bit project the team assumed that a straightforward software migration effort would suffice to move the major part of the software base to the new hardware architecture. In fact, the first plan was for automated migration. Since the move up to a 32-bit system

involved a switch from Intel to ARM processor architectures, the focus for rework was on the hardware interfaces of the low-level system and on the operating system kernel itself, for example to migrate to new process and thread models and a new MMU architecture.

David Wood:

In 1994, we had at first a plan that we would do a quick project to convert from 16-bit to 32-bit. We assumed we would keep more or less the same syntax and that we would actually write a tool to read in everything and then spit out 32-bit versions.

There were a number of reasons for moving from Intel. The Intel x86 architecture was primarily aimed at desktop computers. It had no real power management features, making it power hungry and giving it much worse MIPS per Watt performance than ARM. Just as importantly, even as early as 1994 Psion had identified mobile phones as the most promising future mass market for handheld devices. ARM had established itself as an ambitious and successful semiconductor player in the mobile device market. In contrast, Intel had little or no mobile device presence.

The plan for automated migration did not last long. The language question quickly came to be seen as an important strategic choice, not just a practical one, in much the way that the processor choice was both pragmatic and strategic. One consideration was the nature of the intended platform so far as third-party applications were concerned. The Series 3 had provided a run-time support system similar to Visual Basic for installable third-party applications, and a thriving hobbyist culture had grown up around it. However, this was far from offering a natively programmable platform (which would imply that the system-level APIs were open to third-party applications as native calls rather than serviced through the run-time language support). Native programmability was emerging as a key requirement for the 32-bit system.

The more general question was how far Psion could afford to ignore the mainstream. Continuing with a home-grown system involved more than software maintenance costs (supporting the toolchain and keeping it fit for purpose). It would incur the larger, negative opportunity cost of locking the 32-bit system out of the mainstream. As ambitions for the new operating system scaled up, with its design life projecting well beyond 2000, moving with the mainstream became an increasingly important consideration.

David Wood:

As we saw the scale and longevity of the 32-bit system we thought, 'Wait, we don't want to be stuck in some custom programming system, we want to go

with the mainstream', and increasingly the mainstream was C++. That's what software engineers were learning at university, that's what they would know about. We thought it would be harder to recruit people if they then had to go and learn this comparatively arcane technology which we had. So we adopted C++ for sociological reasons as well as technical reasons. But the technical reasons were important too. As we studied C++ more over the years we saw there were benefits in it being, quotes, *a better C*, rather than just being an object-oriented C.

While not everyone was convinced by this line of argument, it was certainly clear that the alternative to C++ was an in-house system; that C++ was almost certainly going to be one of the dominant object-oriented languages for the foreseeable future; and that the mainstream would be there if it was anywhere. Dialects such as Objective-C were being relegated to the margins by the momentum which was gathering behind C++. Arguably, even Smalltalk, which had some claims to priority, was losing ground to the C++ juggernaut.[2]

While C++ was still the same language which the company had previously rejected as too big, too complex, and insufficiently constraining, the context had changed. Looked at more positively, C++ not only retained many of the advantages of C as a systems programming language (its ability to get 'close to the metal') it was indeed by design 'a better C'.

Stroustrup characterizes this aspect of the language as including stronger type checking; more convenient and less error-prone memory allocation (operator `new`) and release (operator `delete`); function overloading; the ability to initialize variables with values as well as to assign values; references; and, in general, less reliance on exploiting tricks needed to get the best from the language. (Type casting, for example, was a ubiquitous pattern in C that was only rarely necessary in C++ [Stroustrup 1994, p. 171].[3])

[2] A 1995 survey conducted at Object World identified 77% of respondents as C++ users compared with 28% as Smalltalk users [CIO Magazine, ***www.cio.com/archive/110195/object.html***].

[3] It is also worth considering Stroustrup's views on what programming languages are for. A language does two things, as he sees it: it provides a vehicle for specifying actions to be executed and it 'provides a set of concepts for the programmer to use when thinking about what can be done.' [Stroustrup 1993, p. 8] In other words, on the one hand, it provides a model which abstracts a programmable machine (a 'machine model') and, on the other hand, it provides a conceptual model for translating real-world problems into program solutions. Arguably, the history of the evolution of programming languages is the history of the stretching of the gap between the two, so that the conceptual model is increasingly distanced from the underlying machine model, as well as of the increasing abstraction of the machine model away from a literal physical interpretation toward a logical interpretation.

Andrew Thoelke:

I remember hearing about the discussions, but I was too junior in the company really to be directly involved or influence them. But C++ was still quite young at that point. This must have been 1994. It was way off being standardized, probably four or five years before the first pass at standardization. But it clearly had most of the elements of object orientation that had been put into the Series 3 system, and clearly it was going to be standardized, it was going to be much more mainstream, it was important. And the experience of using an object-oriented language and having an event-based application environment was one they didn't want to lose (because the 16-bit system already had that) by migrating to a new hardware architecture and going 32-bit. So it's not surprising they went for it.

Martin Budden, already a veteran at that time, was an early C++ advocate.

Martin Budden:

I was actually advocating C++ fairly early on, but I wasn't the only one, there were other people advocating C++ too. There was a debate about whether to move to C++ or stay with C and, fairly quickly, the C++ argument won. Since Colly was doing the first cuts of the operating system, once he had been convinced, it followed from there.

Geert Bollen arrived at Symbian in the early summer of 1995 with impeccable credentials in object orientation and a significant background with C++, specifically. While others in the company were still learning C++, Bollen already had three to four years heavy-duty implementation experience behind him. By the time he arrived, the decision to go with C++ had already been made and the team was six months into the project.

Geert Bollen:

C++ seemed the obvious choice from my perspective. But I suppose I hadn't really been exposed so much to some of the issues in an operating-system context. But consider, it's only in the last few years that the C++ ABI (Application Binary Interface) has been standardized, so that completely determines the interoperability of different toolsets. That gives you an idea of how young the language was, at least as an industrial tool. All that had been taken for granted in the C world since the '80s; they just weren't issues. In the C++ world they were wide-open problems.

When Bollen joined the project, the lack of standardized low-level tools meant that no code was yet running on ARM hardware, and all development was still emulator-based. The project was still feeling its way, climbing the language-learning curve while throwing its energy into the object-oriented design opportunities of a clean, from-the-ground-up, re-engineered system.

Geert Bollen:

Moving to C++ was a very brave decision. I don't know if it seemed so at the time, but in hindsight it was a very, very brave decision. They had selected C++ to build this system, with some trepidation, as a leap of faith.

Challenges of Switching Languages

Switching incurs costs. There were what David Wood might call the 'sociological' issues, the questions of what tools developers knew, the problems of bringing in new developers familiar with C++ but not with the in-house object-oriented style, while training up the seasoned in-house developers in a new language; and all this, of course, as the design itself evolved fairly freely.

But there were other problems too. Performance problems, perhaps inevitably, quickly became apparent.

Howard Price:

C++ was slow and big in various situations. For example, the OPL VM grew by about six times when it was ported from S3 8086 Assembler to C++. There was some argument, to the point where there was talk of dropping C++ and moving back to the faster, leaner, meaner approach – just sticking to C. But, thankfully, David Wood and Charles Davies were really strongly committed to C++ and object orientation, and they persuaded the rest that this was the way to go.

But the biggest practical headache almost certainly was the lack of a suitable, stable toolchain.

Geert Bollen:

The target tools were not in place when I arrived, so whatever the system was, it was emulator-only. The kernel implementation was an emulator implementation. The on-the-metal version of the kernel was started and delivered after I arrived. Colly Myers assembled a team for that. Before that, he had been a one-man band.

There was no standard toolset capable of targeting both Intel and ARM. Or rather there was, of course, the GNU GCC toolchain, but not one that would work naturally in a Microsoft Windows development environment. Since most development took place on Microsoft Windows for the emulator targets, a productive Microsoft Windows programming environment was essential. Microsoft's compiler and toolchain targeted only Intel processors. Because there was no standard Application Binary Interface (ABI), Intel binaries were in a world apart from the GCC ARM binaries.[4]

In the short term, this meant that bespoke tools had to be written to enable dual-target compile and build using two different compilers, each with its own make and link tools and other utilities, but in a way that integrated reasonably well enough with the Microsoft Windows-based IDE so as not to cripple developer productivity.

The longer-term legacy is that this bespoke system still persists, in essentials unchanged, to this day, a source of low-level discomfort for the external developer community. (Internally, it is as it is and you get used to it. Externally, it is reasonable to ask why you should have to get used to something that causes development pain.)

The underlying problem of course is that phones now, and PDAs then, are essentially embedded devices, and the development process for embedded systems is inherently more complex (typically in arcane and opaque ways) than development for desktop systems. A cross-compilation model is natural and inevitable in an embedded-systems context. For Symbian OS now, as then, initial development is based on an emulator hosted by Microsoft Windows and only later (when the basic implementation is running) moves to a cross-compilation model targeting real hardware, either prototype or production devices or reference boards.

Another lasting legacy of the early tools problems is the nature of the emulator implementation. Essentially, the emulator consists of the user libraries (the base-level services and utilities above the kernel itself) and higher-level kernel services implemented on top of native Microsoft Windows libraries. This is nominally transparent to the higher layers of the system, but key kernel services and hardware-level services are really being provided by the host operating system, in this case Microsoft Windows, to which the low-level calls of the hosted system are mapped. Historically, the problem with this solution is that for anything beneath application-level programming, the true behavior of the code is obscured by the underlying Microsoft Windows implementation. This was not an issue in the early development of the operating system, since the whole point of the emulation was to allow the application developers to get on with work on the application suite, but it rapidly became a problem both for internal and external developers. While the worst problems have

[4] Standardization at the binary level is only now appearing across the toolchain with the standardization and adoption of ARM's EABI, enabling interoperation of tools from different vendors.

been fixed for some time, some awkward issues have remained even up to the most recent releases of the operating system. Those challenges can be traced all the way back to those first problems of early adoption of a pre-standard C++.

As the kernel moved onto real hardware and began to exercise the ARM-targeted toolchain, yet more challenges emerged.

David Wood:

We hit various quite significant obstacles along the way, for example we were calling C++ functions from one DLL into another DLL in a way that I don't think had been widely done before, and certainly this required lots of changes in the GCC compiler that we used. So several times, the implementation for ARM as opposed to the emulator got held up because we were using C++ to the limit, with the DLLs, and we needed to get Cygnus (who were at that stage under contract from us, maintaining GCC) to issue numerous fixes. So we were pushing it to the limit.

Again the issue was the immaturity of the C++ development environment and the lack of standardization of the low-level interfaces.

Yet another area in which early adoption proved to be costly was the lack of standardization at the level of the basic language libraries.

David Wood:

There was no sufficiently agreed standard library that covered the things that we wanted. There was no text-handling library that Colly Myers was satisfied would provide either the necessary degree of security or the necessary degree of efficiency and performance.

This is the reason that descriptors were invented as type-safe, memory-safe, string and binary-data container classes. The alternative would have been to use standard C++/C-subset strings, which provided neither protection against overruns nor type-safety.[5] For a system with pretensions to robustness, they were out of the question.

Error handling was another unstandardized area of the C++ language. Again, a solution was invented, specifically aimed at the needs of the class of device that the operating system targeted, based on the notions of 'leaving' functions and a cleanup stack.[6]

[5] C-style strings are just arrays of raw data bytes, which can be accessed as either character data or raw binary data, and as bytes or multi-bytes. Indeed, that very versatility is their point, but it makes them both error prone and open to accidental or deliberate abuse.

[6] The cleanup stack is used to hold pointers to automatic (i.e., stack-based and, therefore, function-scoped) objects, so that if the function should fail ('leave') the objects are deleted and not left allocated but inaccessible (which is a memory leak).

Other features of the language were treated with caution.

Andrew Thoelke:

We were very cautious in our adoption of advanced C++ features, partly because not all compilers supported them or did them well, but also because you just don't know whether you're using a feature which has lots of power but has a lot of background overhead to it. And we were trying to ensure that we had some control over what was happening behind the scenes. It's still important today, although compilers are better at implementing all of those odds and ends, and having a good relationship with the compiler vendor makes it easier to have confidence in the compiler being efficient.

Charles Davies:

We were experienced in object-oriented programming. We were battle-hardened in object-oriented programming. But, in all honesty, we were still novices in C++.

Managing C++

Beginners' mistakes are the mistakes you make because – well, because you are beginners. C++ is a powerful language and there are almost certainly cases in which its powerful object-oriented features overrode the better judgment of even seasoned developers.

Charles Davies:

We over-used inheritance, which everybody does when they get introduced to object-oriented programming. But we weren't new to object-oriented and should have known better.

David Wood:

We probably did take some of the inheritance hierarchies too far. That was driven by an understandable wish to avoid duplication of code, but some of the hierarchies as a result became, shall I say, unnaturally clever. That cleverness solved the problem of the moment, but ended up itself being difficult to maintain.

Possibly the problem was that because people felt they knew what they were doing with object orientation, they saw in the language only the opportunity to exploit object orientation more fully, rather than the dangers of over-abstraction and over-complexity. However, overuse of inheritance is symptomatic of confusion between class relationships (e.g. the difference between is-a inheritance relationships and has-a

composition relationships) and the misuse of so-called 'implementation' inheritance.[7]

Andrew Thoelke:

C++ is an extremely flexible language because you can ignore all the extras and just use raw C with all of its ability to manipulate hardware and to do whatever you like. But C++ also allows you to be overly excessive with the use of things like inheritance until you get to the point where you lose track of what exactly you are doing anyway.

David Wood:

There's actually a spectrum of reuse. On the one hand, you end up with multiple copies of the same code and that's no good. At the other end of the spectrum, you end up with just a single bit of code which is so convoluted it manages to do everything but no one can understand it. And there's a happy middle ground which nowadays we occupy much more readily.

Certainly there were language features which were considered dangerous, in some cases because they were not well-enough understood in the company (there just was not sufficient grounding in the language for there to be enough experts to make the judgments and provide the design and coding guidelines, short of outright banning), and in other cases because they were not mature, whether in terms of the language specification or of the implementation of the available tools (compiler). Templates, multiple inheritance and namespaces are three examples.

Templates

Templates are a C++ mechanism for writing type-independent code, typically library-like classes that are useful for managing objects of any type, which are then invoked in subsequent code with a parameter of a concrete type, for which the compiler generates (or selects) type-specific code at the point of invocation. They were added to the language particularly to solve the problem of providing useful, generic container classes capable of acting as containers for objects of arbitrary type, in other words, of any type whatsoever that a program might invent (see [Stroustrup 1994, p. 339]). (Ada is another language that provides a template mechanism, though it is different from that of C++.)

[7] 'Implementation inheritance' (in which a derived class inherits its base class implementation, that is, the implementation of the base-class methods) contrasts with 'interface inheritance' (in which the derived class inherits its base-class interface but not the implementation of the base class methods); see [Stroustrup 1993, p. 413].

The problem is that naive use of templates causes code bloat. As a template user, each time you instantiate a given template for a given type, the compiler generates a fresh copy of the complete template code for that type. Each time, every time.

Martin Tasker:

All major systems that existed at that time had been written in C, and C++ was a language that people were still getting comfortable with. There was still a lot of opinion that some of the features of C++ were too expensive to use in a ROM-based small device. Templates being one of them.

Careful design of the initial template classes can avoid the problem, but to do so requires expertise.

Andrew Thoelke:

Certainly parameterized programming using templates is something that's at least a degree harder to understand than some other features. It's very easy to bury yourself in complexity that you assume the compiler will get you out of, but you really don't know. The actual side effects in terms of size of code or performance are hard to measure sometimes. It's hard to write very complex template code well.

Especially in the early days of Symbian's adoption of C++, when the language was fluid and the standard libraries (which were almost wholly template-based) were still evolving and had not been fully standardized, templates were regarded in the company as potentially dangerous and therefore treated with extreme suspicion. [8]

Martin Tasker:

The Standard Library is the basis of what people call generic programming. We do not do that at all in Symbian OS. We have a few templated types, but we don't use them aggressively enough to call it generic programming. I don't think there were many people anywhere that understood the Standard Library at the time it was being designed, and we certainly didn't. Collectively in Symbian, or Psion as it then was, we didn't understand it.

[8] Even the experts find templates hard. See, for example, Scott Meyers in his introduction to [Alexandrescu 2001].

Instead, an in-house idiom called 'thin templates' was evolved for some generic, container-style classes, based around type-independent, pointer-based concrete classes (which therefore can accept pointers to any type of object), wrapped by a template class that contains no implementation, but just serves to enforce a parameterized interface to force type-safety for the object type for which it has been invoked. (The technique is quite well known in the C++ literature, see [Meyers, p. 191, Item 42].) Beyond this, templates are not used in the Symbian OS library classes. If used internally elsewhere, they are used with extreme caution.

Andrew Thoelke:

It was more about the fact that the compilers didn't do a good job with templates. They could only do very basic things. So we were very careful about how we used them. We didn't do anything very advanced. We were also careful to tell people that templates can lead to code bloat if you use them aggressively. So instead we used templates to give type-safety over essentially type-unsafe container objects. Really we were trying to make sure that we could have more maintainable, better written, higher quality code in the first place, by actually constraining the use of the language and helping the external and internal developers to help themselves.

Multiple Inheritance

Inheritance is the basis for structuring the object relationships which underwrite the collaboration and delegation between objects in (class-based) object-oriented design.[9] Inheritance relates objects logically and provides the mechanism for sharing common behavior.

Some object-oriented languages, and C++ is one of them, allow multiple inheritance. (Eiffel, CLOS and Dylan are other examples; languages which do not allow multiple inheritance include Beta and Smalltalk). Multiple inheritance allows objects to have multiple parents. As with templates, casual use of multiple inheritance can lead to multiple instances of identical code being generated. One of the early rules within Psion was therefore 'no multiple inheritance in C++'.

The idea of multiple inheritance is trivial enough to grasp. For example, a police car inherits from Car, but also has Emergency properties, which fire engines and ambulances also share, even though they are not derived from Car (for example, they may be derived from Truck); multiple inheritance allows emergency vehicles to share Emergency properties, inherited from an Emergency class, while deriving from different vehicle base

[9] There are non-class-based systems for example the language Self, based on prototypes (see Chapter 4).

classes.[10] While the principle is simple, the implications for implementation in an object-oriented language go deep.

Martin Tasker:

The multiple inheritance chapter in the Ellis and Stroustrup book is staggeringly difficult! It's mind-bogglingly difficult! So we made a really conscious effort: no multiple inheritance at all. And in fact our solution, Mixins, serves exactly the same purpose as the equivalent in Java, that is, interface classes, and they are really easy for the user.

Andrew Thoelke:

Some of the constraints, like 'avoid multiple inheritance unless the additional base classes are interfaces only', was partly to avoid the Evil Diamond inheritance graph, which you can acquire without always realizing it, and which all the text books said was a Bad Idea, unless you used virtual bases, which we always said, 'No, don't do it', on grounds of performance and footprint and questionable value.

The 'Evil Diamond' pattern is one in which class K derives from both classes B and C, which both derive from A. (K, by multiple inheritance, inherits from both B and C; both B and C inherit, possibly also through multiple inheritance, from A.)

The problem with this pattern is that, because C++ implements inheritance of behavior at compile time not run time, if class A contains concrete method implementations then its code may be and its v-table is compiled into the code for both B and C (because they inherit the behavior) and appears twice in class K.

The immediate problem that outlawing the use of multiple inheritance causes is that no class can present multiple interfaces. An example of the value of interface inheritance is the Observer pattern. Some object has behavior derived through one inheritance hierarchy, say Timer or Alarm, but is also an Observer of some event that triggers its behavior. Not all timers and alarms are Observers, but some certainly would like to be.[11]

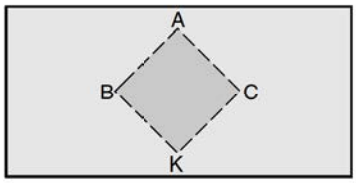


Figure 14.1 The 'Evil Diamond' pattern

[10] The example is from [Stroustrup 1993, p. 405].

[11] [Stroustrup 1994, p. 271] cites Stream I/O as an example of the value of composition of interfaces; another example is composing a class from an implementation and an interface.

Java (as a 'better C++') explicitly provides machinery for adding interfaces to objects. But in the absence of that, multiple inheritance is the most natural way to get it, and the only way to get it by derivation. (So-called 'fat' interfaces are an alternative approach, see the discussion on 'Streams, Stores and Persistence' in Section 4.3.)

Eventually a compromise was reached and 'mixin' ('M' or interface) classes were introduced, following the solution which had actually been first adopted for CLOS.[12] Mixins solve the problem of how to get the best of multiple inheritance, for example, so that objects can present multiple interfaces, without getting the worst of multiple inheritance, code duplication and bloat, and over-complex and over-designed class hierarchies.

With the mixin pattern, while only one inheritance path may inherit behavior, in other words inherit from (and, therefore, include the code from) a concrete base class, a class is allowed to inherit from as many M classes as it likes because M classes may only define pure virtual functions. In other words, they define abstract behavior (interfaces) only, not concrete behavior.

A typical use of an M class is to define an Observer class, so that a `CCoeControl`-derived control (GUI widget) also presents an `MObserver`-derived interface. Because `MObserver` is an abstract class (all of its methods are pure virtual), there is no inherited implementation, and therefore no danger of duplicated implementation code (when you define an `MObserver`-derived class, you must provide all method implementations yourself).

Martin Tasker:

The notion of implementing an interface, which incidentally I think is used very nicely in some of the printer driver classes for instance, seems very natural. So, one pattern for implementing is-a uses mixins; another pattern is that you have an attribute and it turns out that the interface paradigm is actually very natural.

There is a more general and interesting point about the similarities with Java.

Martin Tasker:

We ended up with a lot of the same things that Java did, and we invented them more or less completely independently. We didn't really start looking

[12] Mixins were reputedly named for the way of specifying the flavors you wanted at Steve's Ice Cream shop, near MIT: vanilla plus mixins. The original LispMachine implementation was indeed called Flavors.

at Java until somewhat after our major design decisions were committed. But funnily enough we ended up using multiple inheritance in the same way, we constrained C++ multiple inheritance and ended up using it in the same way as Java uses inheritance interfaces.

Before the mixin solution was adopted, some designs suffered from the injunction against multiple inheritance and had to find other ways to offer multiple interfaces, resulting in convoluted design.

The lesson is that early adoption involves multiple leaps into the unknown. Leaping into the unknown can harm your design.

Namespaces

Namespaces are another example of a useful C++ language feature which was not used because it was not standardized in early C++ implementations. (At least, namespaces were not used early on in Symbian OS.) More recently, some technology areas (networking is an example) have pioneered the use of namespaces, generating some internal debate about what the rules for using namespaces should be.

For example, in designing the networking components, namespaces have proved useful as a mechanism for associating Symbian OS implementations of standards-based behavior (in other words, externally specified APIs) with a standard API, without polluting the global namespace, making it easier for licensees to swap out the Symbian implementation in favor of their own alternative implementations and to do so in principle simply by changing the namespace qualifier.

Freedom Through Object Orientation

One of the more intriguing aspects of the way that object orientation was used inside Psion was to liberate the design process from the need for central oversight and scrutiny, below the level of API definition. While, in some respects, this liberation meant freedom for developers to shoot themselves in the foot with complex features of C++ or just over-use of features (such as over-elaborate use of inheritance), in the wider sense of applying object orientation as an enabling technology to unleash talent to solve a problem while still providing a sufficiently constraining mechanism, it was hugely successful and partly explains how Psion was able to create a fully-fledged modern operating system, all the way up to the application level, in a remarkably short time with a relatively small number of engineers. (In 1997, not long after the first operating system release, Psion Software was still a company with barely 200 employees.)

Martin Tasker:

Something I think Charles Davies did extremely consistently and well, was he basically said, 'Look, you can design an object-oriented system and train all those programmers that you had to recruit because this fantastic system is just too big for your elite team of ten, and you can design the interfaces and give them a sandbox in which to play inside those interfaces, and they can't hurt anybody else by just adding another function onto that library over there', which is what you can do in C, and which is what you are strongly inclined to do in C.

I think Charles, in particular, did this; he paid minute attention to the details of his APIs, he used their explanatory power to motivate his people, and he almost didn't need to look at what they produced in terms of implementation code or test code. He basically said, 'If it meets the requirements of the API and if you feel it's correct then I trust you that it is correct.' Whereas I think that process is harder to do in a non-object-oriented system, because the boundaries are much harder to draw, both for the architect and for the implementer. I think Charles used that to massively good effect. He was outstanding in that.

David Wood was another exponent and practitioner of the principle of freedom and his approach remains to this day: hire talented people, point them at the problem, provide the minimum constraints necessary to bound the solution space, and let them get on with it.

However, this was not what initially attracted him to the benefits of object orientation. Rather, it was the holy grail of reuse. Inheritance as a mechanism for delivering reuse seemed to be a perfect fit for the problem domain of constrained devices.

David Wood:

My first interest in object orientation was in the notion of inheritance, which was that you could reuse somebody else's code but still modify it without having to end up with a complete copy of it. So, the same principle of looking for small code size which runs through many of our early design decisions is there, and we saw that we would otherwise have duplications of code, where there would be a system component whose behavior couldn't be fully parameterized just by data, so you couldn't say, 'well, here are all the flags that will completely govern the behavior of specializations of this object.' We saw that yes, you had to have a way of specializing the code as well as specializing the data. And that is the idea that really struck home. I know exactly the book that I read that made that impact, it was a book by Bertrand Meyer about Eiffel. So that was a very pragmatic consideration, it wasn't at all an ideological consideration.

The object-oriented design approach also seemed to offer something else that was particularly necessary in GUI design: a systematic design approach (abstraction hierarchies) capable of bringing elegance and order to a complex problem domain and empowering designers. Probably the first, large-scale example of a complete object-oriented system (even though it was written in C) was the original Carnegie–Mellon design and implementation of the X-Window system for Unix.[13] Abstraction hierarchies (whether called classes or not) lend themselves elegantly to solving the GUI design problem, while also being open to full-scale extension.

David Wood:

I bought into the wider philosophical view which was that if you followed an object-oriented approach you could handle greater complexity. Because you will have items in your code that map better to concepts in the real world that you're trying to model, or map better to concepts in the user's mind. So you'll have data and code together and you can organize them in hierarchies which correspond to how they are arranged in the real world, or correspond to how they are arranged in the users mind. So, going forwards, that was a very important reason to adopt object-oriented principles. It allowed a single programmer to hold more in his head.

Again, the efficiencies delivered were not just machine efficiencies (more effective, efficient code) but human ('sociological') ones too: more productive, effective and empowered developers.

David Wood:

You always fought against code bloat, and there are two ways to fight it. The first way is to encourage people to practice efficiency in the small, which was to understand the side effects of the code they were writing and to think about every line of code they wrote to make sure it wasn't unnecessarily long and they weren't duplicating code unnecessarily. The second way is to look for efficiency in the large scale as well, which is when you realize that two programs, or more than two, are actually trying to do essentially the same job, so instead of them having independent copies of code which is largely the same, you have one copy of the code, together with small bits in each of the applications where they provide their relevant specializations.

Object orientation provides a principled approach to code reuse. In contrast, nothing is easier or more tempting or more fatal, than reusing code by literally copying and pasting code sections within programs and

[13] At the time of writing, there is a good history at ***http://en.wikipedia.org/wiki/X_windows.***

between programs; object orientation is the ultimate principled antidote to 'copy and tweak'.

David Wood:

What we were fighting against was something that's sometimes called 'copy and tweak', when you're writing some code and you notice that something you've already written doesn't quite do the job but it nearly does, so you just copy it wholesale and then tweak it, so you end up with two not quite identical copies of the code and before you know it you end up with dozens of copies of essentially the same code, but it's not quite the same. And that's bad for all kinds of reasons. It's bad because it's inefficient. It's bad because it's very hard to maintain, if you discover that there's a problem with your original algorithm you may manage to fix it in one place and then the same bug remains very probably in all the dozen separate copies that you made in the meanwhile. Some people nowadays say, 'Well, the hardware has advanced so much, the same efficiency considerations don't apply.' And there's certainly an element of truth in that. We needn't work quite so hard to save every single byte. But avoiding duplication is still an important principle, because if you duplicate unnecessarily then you end up with maintenance problems and comprehensibility problems.

14.3 A Thoroughly Object-oriented Operating System

The rest of this chapter looks in more detail at how object orientation is used in practice in Symbian OS, identifying some of the most characteristic examples of object-oriented style and techniques. Symbian OS contains some important object-oriented patterns. What follows is not an exhaustive list, but it captures the larger scale object-oriented patterns in the system.

- Frameworks and plug-ins are a good example of the power of polymorphism and are ubiquitous in Symbian OS. As a pattern, frameworks operate at the level of the static architecture of the system (what parts are in the system, in other words) although they also have an interesting run-time, dynamic aspect.
- Active objects are thoroughly object-oriented and are probably best thought of as a design and programming idiom for avoiding the complexity of multithreadedness, while enabling asynchronous activities to be spun out of a single thread and spun back in.
- Descriptors are good examples of object-oriented encapsulation, best thought of as an idiom for achieving type-safe and memory-safe string handling.

- Cleanup is another example of good object-oriented design applied to solve the error recovery and propagation problem and is probably best thought of as a programming idiom (although it also touches the built-in typing model).
- Streams and Stores (the persistence model, in other words) is pure object-oriented design, that underlies a programming idiom for externalizing and internalizing data through an abstract, non-file-based abstraction.
- The User Library is a good example of simple encapsulation-style object-oriented design used to package some system services.

On the other hand, there are also some designs in the system which are more conventional, non-object-oriented patterns.

- The client–server model is not an object-oriented model (unlike frameworks, for example), even if it is given an object-oriented flavor by modeling the client–server relations with objects.
- The file system, beneath the client–server and framework architecture and the object-oriented interfaces provided by the stream–store persistence model, is in fact a conventional, non-object-oriented, files-on-disk system. (Compare for example with the thoroughly object-oriented 'object soup' storage model made famous by Apple's Newton and the similar database approach later adopted by Palm.)
- Critical aspects of the kernel design, for example the scheduler, are conventional in design, with hand-optimized assembler implementations for speed.

It is worth making the observation that the device for which the operating system was originally designed, the Psion Series 5, was quite a conventional device in terms of its technologies. So while the use of the technologies was innovative, the technologies themselves were conventional: a small, ROM-based device with screen and keyboard using conventional, removable (Compact Flash) data cards for external storage. To take the file system as an example, the choice of a FAT filing system was conventional but eminently reasonable. And in fact, if users wanted to use their removable cards to swap data with other devices, then FAT was the necessary choice, for removable media at least. Deciding for a conventional file system simply reflected the realities of data exchange for a handheld device in a world dominated by PC consumer computing.

This argument in favor of a conventional file system beneath the object-oriented wrapping is one that Geert Bollen, for example, has heard before.

Geert Bollen:

That's very much the argument you would have expected to hear from Colly Myers. But, it doesn't follow from the argument that the internal storage representation should be FAT, though that's quite a valid design choice.

The point is really that, while the system indeed is thoroughly object-oriented, adoption of object orientation was motivated by pragmatism and, in some cases, that same pragmatism led to more conventional, non-object-oriented design choices in the system.

Frameworks

A framework is a component that defines an abstract interface which it does not implement, but for which instead it provides a runtime loading and management mechanism for locating and loading external plug-in components that provide concrete implementations for its interface.

Framework–plug-in is a classic pattern therefore for separating interfaces from implementation, which is one of the driving design principles of object orientation (inherited, as it were, from the notion of the abstract data type, an important influence in the emergence of object-oriented languages). The framework–plug-in pattern is therefore classically good object orientation. It is also a natural pattern for enabling extensibility.

But frameworks are more than that. As [Beck 1999, p. 258] puts it, 'Design is hard'! A framework is, in effect, reusable design because it expresses a part of a system as a set of abstract, cooperating classes, scopes the behavior of those classes by defining their interfaces, and implements the interface between the framework and the underlying system; in other words, it wires the design into the system. At that point, the design work is complete. What remains is implementation, and implementation (the actual instantiation and also, if you like, the interpretation of the behavior scoped by the interfaces) is left to others including third parties.[14] Frameworks are an important design choice, because a framework design strongly determines how we should expect to work with some part of the system. In the object-oriented literature, frameworks go back to ideas of Deutsch and Johnson in the late 1980s.[15]

Of course, patterns are also 'designs for designs' and, arguably, so too are class libraries, but frameworks go further than patterns, because they

[14] Or as Martin Budden put it to me, 'Frameworks are for people who just can't make up their darned minds!'

[15] My references here are taken from [Beck 1999, p. 257].

are actually coded ([Rising 1998, p. 375]), and they go further than class libraries, because they are already pre-wired into the system.[16]

There is an objection: frameworks are complex and hard to write and need 'refining by fire' (see [Rising 1998, p. 184]). In many cases there is an additional, important dimension. Where frameworks expose interfaces on two sides, an 'up-side' application-level interface and a 'down-side' hardware-level interface (which is a common pattern in Symbian OS – the frameworks in the communications and multimedia areas are good examples), the question of the 'thickness' of the API, (how much code there is between the up-side and down-side interfaces) can become critical for retaining control of the design. The thinner the API, the more exposed it is to the opposing and even hostile forces of its up-side and down-side clients and the easier it is for them to subvert the original design intention.[17]

Frameworks can also be complex to use. Because they constrain the user, they must be understood well to be used well. Frameworks are a pervasive design choice in Symbian OS, because they are a particularly effective mechanism for enabling customization and extension at a deep level; the operating system implements an overall design in some particular area but licensees are still able to contribute highly customized behavior. Frameworks therefore can be found in all layers of the system, from top to bottom. Indeed, in some senses, the different ways that frameworks are used in the different layers may be the dominant design characteristic of those layers.

Model–View–Controller

Model–view–controller, or MVC, is a well-known design pattern which applies the idea of frameworks to the design of an application, 'the earliest framework that was recognized to be a framework', indeed, according to Johnson [Rising 1998, p. 376].

The high-level design of an application in Symbian OS really resides in this pattern, which provides a general model for the classes that an application needs, what their basic collaborations are, and how they are achieved.

The simple version of MVC, and the headline from the point of view of Symbian OS, is the separation of the application model (the data and the APIs which operate on it) from the application logic (the way the user uses the application and its data), not just as a coding rule, but as a well-founded design principle.

---

[16] As Johnson puts it, the difference between patterns and frameworks is the difference between reusing knowledge and reusing code. Frameworks reuse code and, thus, enable more immediate reuse than patterns [Rising 1998, p. 382]. Patterns are a degree of abstraction further out from frameworks.

[17] See Chapter 16 for some examples of the 'thin framework' problem.

Howard Price:

Model–view–controller goes way back to SIBO. It probably was Charles Davies and David Wood who introduced it. But, right from the beginning, that's always been the design. Right from the beginning, there was a decision that you really should separate these three things.

Active Objects and the Event Model

The event model is an important part of the wider application framework, but it is itself a part of the system which makes rich use of object-oriented principles. Its key principle is to abstract the event loop using the active object pattern. Again, the driving point here is to fix the design in the framework and so simplify the way that applications interact with it through the framework.

Martin Tasker:

Think of it this way. Firstly, the fundamental requirements of doing a GUI and the fundamental requirements of event handling motivate object orientation, and polymorphism in particular. So you have an event and you have a control, you send that event to that control or you get that event from that control. But then of course you need concrete types underneath that, you need concrete actions for concrete controls and concrete events. The point is that the fundamentals of event-orientated programs in a GUI context take you in a very short sequence of steps to object orientation.

At the heart of the message dispatch system, active objects are used to encapsulate and serialize (i.e., make sequential) incoming events. In Symbian OS, this is probably the object-oriented pattern which goes back the furthest. The alternative, of course, is the conventional 'big switch' statement style of classic Microsoft Windows.

Active objects provide a simple, natural, serialized, alternative.

Martin Tasker:

Colly Myers was right, active objects are a dream. For people who know that they are dealing with event-handling programs, they're an absolute dream. In an event-handling system, active objects are a really natural way of handling things and they are so much easier for programmers to work with than pretty much all of the alternatives actually, or any alternative that I had seen.

The key advantage of active objects is that they provide a simple model for maintaining a single flow of execution through the main

program logic of an application, while still enabling a responsive, event-based implementation. They offer a simple alternative for what would otherwise typically require a complex, multithreaded approach.

Martin Tasker:

With multithreaded programs you have to do locks, so you always have to ask yourself, 'Am I handling these locks correctly?' And what eventually happens in those systems is that you get some conservatism, so you eventually end up with two or three global system locks which only your kind of super-elite programmers ever touch, and then you get a couple of local locks which are more or less private business, and you don't quite have to be a genius to do stuff with those locks on the basis that you only penalize your own code, so you're probably going to get it right eventually. But it's much more complex and it's much less elegant.

There are times, of course, when the active-object model is inferior to a more raw-thread-based model.

Andrew Thoelke:

Of course there are times when the active scheduler gets in the way, but for many things active objects are a better model. It's actually a better model to have cooperative active objects running in a single thread than to go multithreaded. But there are certain tasks for which multithreaded programming is definitely more useful. Certainly when you want to be able to do finer time slicing or you don't want to have to manually break up a task into discrete chunks. Sometimes it has been hard initially to integrate that sort of software with Symbian OS. In the early days, a client–server session was inherently tied to a thread. The client was the thread and that was engrained in the whole framework and model. And there were even cases where that was designed into the server. The server required that the thread and client were synonymous. That has changed now, explicitly to enable multithreaded clients. So you have the choice.

The Framework for Frameworks

An important change in Symbian OS v7.0 was the introduction of the ECOM 'framework for framework plug-ins', the Plug-in Framework component.

Andrew Thoelke:

The plug-in pattern was pervasive in the OS. Everything has plug-ins: the kernel does, the file server does, the window server does, and in all cases the actual

mechanism for tagging and identifying and locating a suitable plug-in that matched your interface protocol and statically matched your actual specific requirements right now, was very ad hoc, except for the fact that you generally used a UID to match the server protocol interface. But the way you located them, whether you searched for them, whether the loader searched for them, it was all very unpredictable.

Worse still, new variations were continually being created, of varying quality. Standardizing framework–plug-in design by introducing a framework for framework plug-ins not only brought more discipline to bear on the way plug-ins were searched for, found and loaded, it allowed the best design (the plug-in system created for the web and Internet browser) to become the template.

The basic principle adopted is the abstract factory design pattern.

Andrew Thoelke:

Our design is a fairly obvious pattern as soon as you see it. You can see that actually I can apply this anywhere somebody wants an abstract factory. You put that abstract factory something in a DLL, and then when you request that DLL you have it conform to your interface. And the ECOM plug-in framework just provides a standard way of doing all of that.

As it happened, standardizing the plug-in mechanism fit in well with the new requirements of the platform security architecture. (Unchecked loading of externally written plug-ins into system frameworks, or applications for that matter, is a potential security risk.) And that in turn helped to enforce the standardization, because it could be pushed through as part of the effort to implement the system-wide, pervasive changes required by the security architecture.

Andrew Thoelke:

With platform security, it was highly desirable to get away from everybody searching for their executable content, which is what's really going on when you're looking for plug-ins to load into your framework. And because we already had put ECOM there and it was an established part of the system, we could enforce it as the standard mechanism and police plug-in loading at that single point. So we had a way to force the migration to ECOM through. After PlatSec, there is no way for your framework to find out what plug-ins are available by enumerating the actual executables in the system because it's not possible to do that any more. So you either use ECOM, because it's got its own registration mechanism with resource files, which regulates add-in plug-ins, or you have your own registration mechanism where something which gets

installed not only installs the executable, it installs something else which tells you that there's an executable plug-in that you can load, so you control what plug-ins can be added.

Streams, Stores and Persistence

Symbian OS, possibly surprisingly for an object-oriented system, supports a conventional FAT file system. At the same time it provides a native, object-oriented persistence framework that, for example, enables applications to externalize and internalize state without explicitly invoking file system APIs. It also provides a native database storage model, which is extensively used by applications such as Agenda and Contacts. Geert Bollen was the original architect of the persistent-storage frameworks.

Geert Bollen:

At my previous company I had implemented a persistent object database over RDBMS in C++. Actually, I had done that several times. So I was brought in as a C++ and design expert, and I was given the persistent-storage job. I had a team of two, me and Andrew Thoelke.

Andrew Thoelke:

It was only 18 months after I started and EPOC 32 was really kicking off in a big way. So I started working with Geert Bollen shortly after they had the major discussion about 'Is it going to be a database? Is it going to be a file system? Is it going to be some object soup? Are we going to buy it in? Are we going to write it ourselves?'

To start with they designed and implemented a DBMS API based on the DBMS API in the 16-bit system (which had originally been written by Colly Myers).

Geert Bollen:

The design and implementation is that of a classic, by-the-book RDBMS light, because it was always designed, for example, not to have joins. It was light.

The design layered a stream serialization interface over a dictionary-style persistent store over a (conventional FAT-style) file system.

Andrew Thoelke:

Initially they were going to say, 'Right, well pretty much every file's going to

be a database' and that was refined a bit later on, so it turned out that every file should be a database but they were going to go with a file system, and you had different sorts of databases for different sorts of files. And then the ideas got refined somewhat to having two separate layers, having a store architecture, and the notion of Streams and Stores which is slightly more basic than a full database, and then you have the database sitting on top of that for applications that explicitly want that kind of data storage.

Geert Bollen:

DBMS was always intended to be the heart of application storage. The Psion Series 5, recall, was to ship with a suite of built-in applications, including Agenda, which was a diary, meetings, and to-do list application, as well as office style apps. DBMS was intended to provide storage for them all.

For more conventionally file-based applications, such as the spreadsheet and the word processor in the original Psion Series 5 application suite, a more document-centric Store interface was available.

Later, the decision was taken to design a less abstract storage framework with object serialization at the heart of it. DBMS is a natural storage model for a database application, such as Data, but, for document-based applications, the requirement was somewhat different.

Geert Bollen:

We needed a Store for document-based applications which load up their application data when they launch and write it out when they suspend.

The question was then how this fit with the needs of the more transaction-based applications (such as the Data application, which explicitly used a database format, or the Agenda application, which implicitly used a database format); in other words, applications which didn't load their data into working memory, but had a store on permanent storage and conducted transactions against the store.

Geert Bollen:

The requirement was to build a framework which combined the needs of database operations with the simplicity of object serialization. That led to a transaction-based API for Store and we provided that as a fat API-style framework.

It is a 'fat API' because its single interface had to encompass the multiple interfaces it needed to support the multiple different kinds of

objects. In other words, Store exposed a single, 'maximal' interface which was only partially implemented by any one of the concrete objects which implemented the Store interface and which were available at run time. In other words, the concrete objects which were available at run time each implemented a subset of the complete interface. The 'fat API' approach was adopted as an alternative to multiple inheritance, against which there was a strict injunction. Since at that time the 'mixin' solution had not yet been adopted, there was no other way to expose multiple interfaces.

Just as the database layers over the generic Stream–Store architecture, so it in turn layers over the more or less conventional filing system.

The goal of the Stream–Store design was to define a generic persistent storage mechanism suitable for any application type and robust enough to guarantee bullet-proof data safety in a model in which 'the user never saves'. Data safety was required no matter what the user might do or fail to do, including pulling out a removable media card while an application was using the data stored on it or even pulling out the batteries while applications were active.

Peter Jackson:

The end-user requirement is that you don't want corrupt data still around, you want a transaction-oriented filing system at some level so that if something goes wrong in the middle of what you're doing you don't have to do some expensive repair process. You might have lost a couple of transactions because that's when it went wrong, but the transactions that have already happened are safe. And the whole Stream–Store technology gives you that kind of layer. And for that reason I would say it's a lot better than just having a raw I/O system.

Andrew Thoelke:

I think for Streams, in particular, the actual class design went through something like four or five iterations, because we were trying to deal with the many ways you might want to use Streams and chain them together and run them back to back without resulting in a system that passes one byte around between different classes, which gives you performance problems.

Store is a complex design problem because what's required is really not a single solution at all, but multiple orthogonal solutions, serving the fundamentally different needs of different applications working with data in essentially different ways, that is, document-based versus transaction-based.

Andrew Thoelke:

DBMS is just a template API for a database, whereas Store is an extensible

Steams and Store architecture where you can have different types of back end to it.

No design allows an application to continue writing to or reading from a media card that the user has removed from the device. However, different design choices provide different levels of protection against such unexpected failures. A conventional file system is the least robust solution. From a developer perspective, on the other hand, file-based semantics are so widespread and so engrained as to seem like second nature. In contrast, object-based serialization using externalize–internalize and store–restore semantics imposes a new learning curve at what should be a very basic level of programming, saving and restoring data. Inevitably too, if developers don't understand a model, they use it wrongly (or, indeed, even deliberately subvert it). In some respects then, the persistence architecture is another example of a Symbian OS idiom which has been described as a barrier to new developers, but which is firmly rooted in the original design requirements for the system (unrivalled robustness for a device class which is quite different from the conventional desktop device).

Object Orientation in the Kernel

The Application Architecture framework, the Control Environment hierarchy (CONE), the Graphics Device Interface framework and Store all make aggressive use of object-oriented and C++ techniques including interface inheritance, polymorphism, templates (in the Symbian OS 'thin template' style) and so on.

At first sight, most of these techniques are absent in the design and implementation of the kernel (whether the EKA1 or EKA2 architecture).

Andrew Thoelke:

If you look at the features of C++ that get used in the kernel, you don't see very much in the way of templates and you don't see very much in the way of derivation and you don't see very much in the way of virtual functions and overrides and frameworks, so it's a simpler use of object-oriented design. But it's still object-oriented design.

In fact, the kernel design has some interesting object-oriented features and they are sufficiently fundamental to persist in the design of EKA2.

Geert Bollen:

There is something interesting. There is a key insight of Colly Myers's which is very interesting, and it's an interesting use of OO concepts. A lot of what an OS does is very close to the metal, which is not an area that lends itself to standard OO-programming mechanisms. But Colly's key insight was that at one level the kernel is a model of an executing system. That's what the kernel's job is, parceling out the hardware resources of the underlying machine, and of course managing the computation, which is what the central processor is there to do. So what else really is the kernel? The kernel is really just a dynamic model of the computation which is in progress, of the thread of execution and the processes which exist and the way they are locked and communicating. You can make an OO model of that. You can implement that model using a bunch of objects. And essentially that's what Colly did. That was a very interesting use of OO concepts to design a kernel.

The basic kernel objects derive from this approach.

Andrew Thoelke:

Any kernel is going to have a control block for a process, but rather than just saying it's a control block, in Symbian OS it's actually an object with responsibilities to manage that process and the things that belong to that process. So it's not just a data structure that says, 'This is the process control block associated with this dynamic collection of threads and memory space.' It's an object and it has the responsibilities for that management within the kernel.

The kernel takes a different approach to the use of object orientation than is taken in the UI or Application Services layers, for example. Nonetheless the 'object-oriented clouds' are there.

Andrew Thoelke:

In the kernel in some sense you can say there are clouds, because you've got encapsulation, and you've got objects. Perhaps it depends on whether you say, 'Well, if you don't really use advanced inheritance and virtual functions then you aren't really using OO', right? So OO is derivation, using virtual functions, polymorphism and whatever, but a key part of it is about encapsulation and roles and responsibilities and the way the different objects in your system interact with each other using defined methods or messages, rather than by just invasively tinkering with each other's data structures. That's OO too. You can see that in the kernel.

Object orientation is not just a way of making a design tidier. It enforces discipline, not just on the design of a piece of software, but also on the way that users must go about designing their own use of the software.

Andrew Thoelke:

From the point of view of modeling a system and having clearly defined components which interact in well-defined ways, the kernel is really quite object-oriented. Whether EKA2 is as rigorously so, it probably is at the same level although for the sake of performance occasionally we don't always try and protect data. But from the point of view of saying, 'Are we utilizing virtual functions and polymorphism substantially?' – well the kernel has got one or two examples of it, the key one really is the device-driver framework, where you have a base class which represents the basic device driver and device driver-channel, and then a real device driver which implements a concrete device-driver object and device-driver channel. But, on the whole, that isn't really used in the rest of the kernel, for the good reason that the rest of the kernel is a closed system, and there is not a great deal of value to be added by pursuing that kind of design extensively.

There are some other choices that Colly Myers made which are interesting.

Geert Bollen:

Colly Myers decided to use polymorphism for implementing the differences between the target machine and emulator implementations. For example, `DThread` is the abstract base class and you have a derived emulator thread class and a derived on-the-metal thread class. I'm not so convinced about that.

Bollen's objection is that polymorphism in this case is the wrong concept to apply.

Geert Bollen:

I'm not convinced, because it's using a dynamic binding mechanism, that is, polymorphism, to represent a static property of the system, that is, whether you are an emulator build of the system or an on-the-metal build of the system. You don't need to do that at run time. In fact, you really don't want to do that at run time. That could as well be a compile-time or link-time binding.

Using a complex mechanism where a simple one would do (as Bollen says, 'A typedef would do it: `#ifdef baretothemetalthread...`') seems to break a basic design principle, though no doubt there was a

reason for the decision when it was made. But there is a deeper point, echoing one previously made.

Geert Bollen:

Modeling the computation, making that literal and the operations that you do is still an interesting choice. And it points out something else which was going on, the desire in the architects to constrain the design choices, to have a limited number of patterns in the system. This is an OO system, so let's use OO mechanisms. There's some possible impact on runtime? So?

15

Just Add Phone

15.1 Introduction

Mobile phones are uniquely complex devices: more complex than PDAs; more complex by far than PCs. This case study looks back to the critical early points in the evolution of Symbian OS into a fit-for-phones operating system. It looks at why phones are different, what particular challenges they pose and the impact of those challenges in shaping Symbian OS.

It is often said that Symbian OS was 'built for mobile phones' and the claim, while true, does not remind engineers of the roots of the operating system. Its predecessor, EPOC, was conceived and implemented first as a mobile operating system for PDAs, even though, by the time it first shipped, mobile telephony had been identified as a critical market in which the operating system would win business and work was well in hand – driven by phone licensee collaborations – to put Symbian OS inside phone handsets.

As this case study shows, the shift in emphasis from PDAs to phones involved some very real challenges. Understanding what those challenges were helps us to build a deeper understanding of what Symbian OS is today.

15.2 Anatomy of a Phone

'Mobile phones', says David Wood, 'are just irreducibly complex.' He should know. As one of the five founding directors when Symbian was created,[1] Wood took responsibility for the Technical Consulting arm of the company – with a mission to support licensees, old and new, to create 'great phones' on Symbian OS. Technical Consulting has always been at the forefront of the process of securing licensees' use of the system and following through to shipped products.

[1] The others were Bill Batchelor, Colly Myers, Stephen Randall and Stephen Williams.

David Wood:

There's no getting away from the fact, it's not Symbian OS that makes smartphones complex. Smartphones are complex simply because of the enormous number of different technologies that are contained in every single smartphone.

Compared with PCs or even PDAs, phones pack an astonishing number of different technologies into a tiny package. The things which in a PC are peripheral become integral in any pocket device; screen, keyboard, speakers, microphone and soundcard are packed in with CPU core and memory and permanent storage. But phones, of course, go further. There is the phone radio hardware itself and possibly multiple other radio interfaces, such as Bluetooth, and their associated software stacks, full networking, and a complete multimedia system. Megapixel cameras with true optics have arrived (Zeiss lenses, for example, and optical zoom) as, of course, has stereo sound with real-time compression and decompression for stereo playback. Integrated high-definition TV is the most recent arrival, hard on the heels of Wi-Fi, with Wi-Max waiting in the wings. Add in the power-management technologies needed to deliver long battery life while fueling this impressive array of technologies with ever-increasing processor speeds yet also avoiding the device becoming (literally) hot in the user's pocket; the PC, in comparison, starts to look trivial.

Looking back, the hardware architecture of the early devices for which Symbian OS was originally designed now looks remarkably simple too. A hardware schematic of the Psion Series 5[2] shows little more than an ARM core connected via a data bus to ROM, DRAM, and removable media-card memory, with direct connections to the remaining hardware: an audio codec for microphone input and speaker output, RS-232 and infrared UARTs, LCD screen and digitizer overlay (via an analog-to-digital converter) and keyboard (via parallel I/O pins). A power supply unit drives the system from two standard AA batteries and a wristwatch-style flat cell backup battery. Interestingly, the Series 5 hardware was itself more complex than that of some competitors, for example, Apple's Newton or Motorola's Magic Cap devices [Wolf 2001, p. 548].

15.3 The Phone Operating System

A phone operating system must manage complexity at a number of levels.

- It must manage the sheer hardware complexity of a converged device but, more specifically, it must manage a double platform: a highly specialized, data-centric (including voice data[3]) radio hardware device on

[2] [Furber 2000, p. 366] shows just such a schematic.
[3] Voice-centric GSM/2G has morphed into data-centric 3G.

the one hand and a multimedia-ready, networked, application-centric device on the other.

- It must cope with the sheer software complexity and, more specifically, a double software stack: specialized communications and data-centric protocol stacks and real-time channels on the one hand and a GUI-based, friendly, application-rich consumer system on the other.
- It must deliver the performance expectations of a general consumer market (toasters 'just work' and so phones had better 'just work': when they stop working, they are thrown away), quite different from the expectations of users of desktop computers, PDAs or gadgets.
- It must conform to the usability expectations of a general consumer market (no one expects to read the manual for a toaster; why should they read the manual for a phone?), again quite different from specialist users.
- It must stay fit and keep up to date with rapidly evolving technologies, a rapidly evolving network-services (operator-services) infrastructure and evolving open standards (often multiple, competing global standards).

Depending on who you talk to, 'phonification' means different things but, in principle, it means all of the above. 'Being a phone' is different from being merely a small, pocketable, mobile device.

Architectural Impacts

The impacts on the system architecture of the move from PDAs to phones are worth examining.

- The most obvious impacts are on the Kernel Services and Hardware Interface layer. The Kernel itself must either support the real-time needs of the baseband or it must be made amenable to an alternative solution. It must also support licensee ASIC or ASSP custom chip packages, which may mean supporting alternative memory models and memory-hardware architectures. It must support phone-specific device peripherals, including screens and keypads and possibly dedicated hardware such as a phone flip.
- In the Base Services layer, the File Server must support multiple file-system architectures and media types (both NOR and NAND flash, for example, and probably also hard drives), providing specialist services such as wear-leveling (for NAND) and demand paging.
- In the OS Services layer, the Comms Services architecture must support telephony protocols and integrate with networking support; graphics and multimedia services must support phone use cases (such as

camera-phones and music-player-phones); connectivity requirements are probably also different between PDAs and phones.

- The Application Services layer is significantly affected too, with the different phone use cases for applications and new services requiring support.
- Finally, the UI Framework must provide support for a dedicated phone user interface.

There are almost certainly other system-wide impacts:

- system performance characteristics are likely to be quite different for a phone and a PDA
- greater modularity in the system may be necessary to enable the different product cycle for phones (a more platformized model, in other words) and a full product matrix
- adaptation almost certainly becomes more complex, with a much greater range of technologies needing integration (web browsers, viewers for content such as Flash and PDF, and Office file viewers, font technologies, etc.) and plumbing to close hardware support
- service assumptions turn into architecture headaches; phone operators are simply not used to open network models (third-party software availability, for example) and a system-wide security model becomes a requirement.

Supporting the Baseband

The mobile phone 'baseband' is the software signaling stack that accepts a data stream (e.g., the output from a digital signal processor which has been fed voice input from a microphone) at the top layer and emits a stream of encoded data frames at the bottom layer onto a data channel to radio–air-interface hardware and vice versa; accepting encoded data frames at the bottom and emitting a data stream at the top (if it is voice data, it is fed to a DAC for conversion to audio output).

The signaling stack sits on a software layer of some kind, traditionally a real-time operating system (RTOS), which in turn drives the baseband processor, typically a general-purpose CPU such as an ARM or StrongARM dedicated to the RTOS and the stack that it hosts. The radio hardware is likely to vary between phone makers, from custom silicon to standard bought-in parts, and the data channel might be old-fashioned serial or, more recently, USB.

Non-voice data on 2G and 2.5G and all data on 3G is packetized (into TCP/IP packets). The DSP/DAC steps are omitted but otherwise the

packets follow the same path through the signaling stack and are tunneled as GSM/2.5G/3G frames.

The basic hardware design choices for a Symbian phone revolve around whether to use one processor or two and, if two, how to connect them. The two-CPU option treats the baseband, or phone side, as a completely separate hardware subsystem and interfaces it to an application subsystem with its own application processor (an ARM or StrongARM CPU, for example), running Symbian OS and applications. The single-CPU option creates a single hardware system, and shares it between an RTOS and Symbian OS or (enabled by the Symbian real-time EKA2 kernel) uses Symbian OS exclusively to host both the baseband and the application stack on a single CPU.

While to some extent how licensees architect their phones around Symbian OS and the design choices they make are opaque (and often jealously guarded), the consequences clearly impact the operating system design and the assumptions it makes about the environment in which it runs.

The hardware design options are as shown in Figure 15.1.

A: Two processors connected by a fast serial bus: CPU A is the baseband processor and hosts an RTOS, which in turn hosts the baseband stack. CPU B is the application processor and hosts Symbian OS, on top of which is layered a bespoke or licensed user interface that hosts applications.

B: A custom package with two CPUs and shared memory at the register level. CPU A is the baseband processor and hosts an RTOS, which in turn hosts the baseband stack. CPU B is the application processor and hosts Symbian OS, on top of which is layered a bespoke or licensed user interface that hosts applications.

C: A single processor hosts both the RTOS and Symbian OS. RTOS runs the baseband stack and Symbian OS runs the user-side processes; the two operating systems have a mutual agreement to share the CPU, RAM, device drivers, and other system resources.

D: A single processor hosts Symbian OS with real-time kernel EKA2. Symbian OS, abstracted by a custom 'personality layer', runs the baseband stack, device drivers and user-side application processes.

But telephony is not just a matter of getting raw data to the baseband. The baseband needs to be under application control, which means that there must be application interfaces to the phone side from the application side. On a typical Symbian phone, the phone application is simple and can therefore be relatively hard-wired to the phone side, but phone

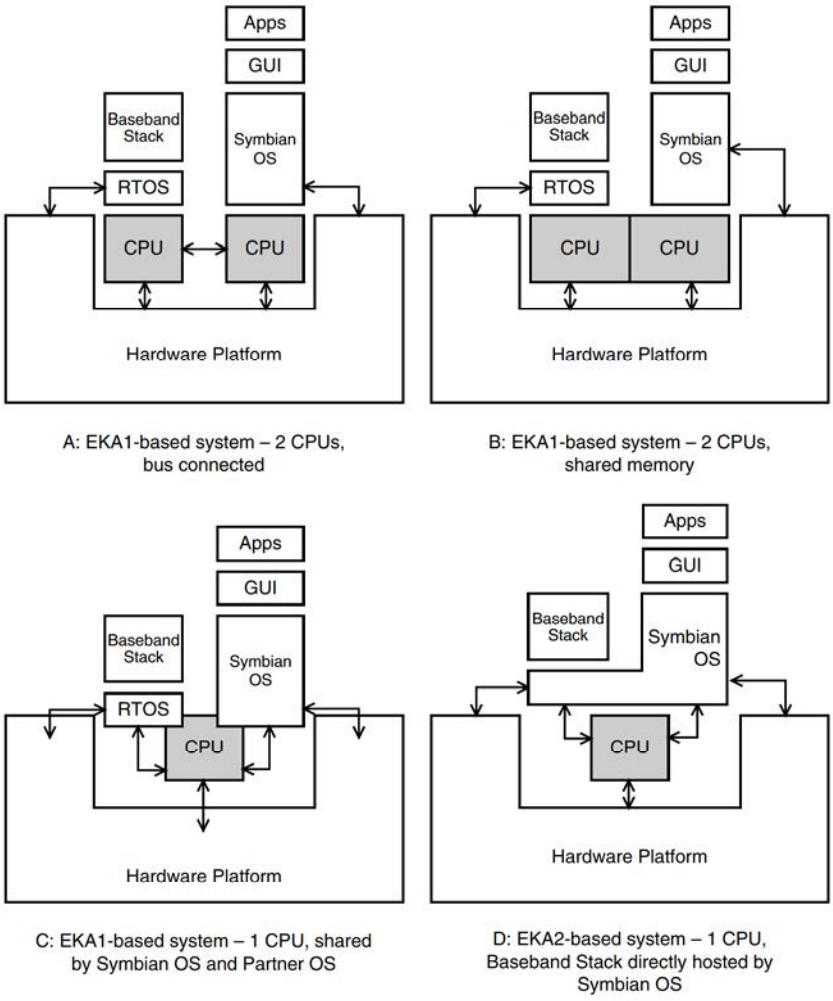


Figure 15.1 Hardware design options

books, call logs, messaging (SMS, MMS, email, and so on) require access to phone protocols and data services (such as networking and Web browsing) need to control the phone side in a modem-like fashion.

While the telephony application that the user sees is relatively simple, the underlying engine which sits beneath it is quite complex. It needs to handle a number of cases such as ensuring that emergency calls are always possible, even in low-memory conditions, and it must interoperate with hardware accessories such as headsets, as well as specific call-handling and over-the-air (OTA) settings protocols (e.g., call-handling and SIM toolkit functions).

Supporting New Hardware

The real-time kernel (EKA2) is still valuable even in designs that retain a separate, dedicated RTOS or partner operating system. Whether or not it hosts the baseband, EKA2 has advantages on the application side too.

> **Ian Hutton:**
>
> Hosting the telephony stack directly on Symbian OS requires real-time capability – and this is an issue. But the other argument for EKA2 is that it will allow mobile-phone manufacturers to integrate more multimedia hardware. The increased complexity of the hardware puts demands on the operating system that are increasingly hard to sustain, just with the level of interrupts and so on, and that's where EKA2 makes the difference. So enabling the integration of more and more hardware without compromising performance, which is what we are witnessing with phones, is the real bonus.

The other aspect of new hardware – cameras, audio-codecs, high resolution displays and multiple displays, multiple radio interfaces and new memory formats such as NAND flash, to give the most obvious examples – is that it needs drivers and, increasingly (and especially for more exotic hardware), the drivers are likely to be proprietary rather than supplied by the operating system. Easy integration of third-party and partner drivers becomes a significant matter. EKA2 has been designed with these needs in mind.

Supporting Services

Networks are driven by services and services are supported by phones. As voice services become increasingly commoditized, existing non-voice services such as messaging, browsing and new services (alternative network access through Wi-Fi, broadcast TV, presence and navigation) become more important.

Supporting Features

There is no good definition of what features make a smartphone into a smartphone, a mid-range phone into a mid-range phone or a low-end phone into a low-end phone.[4] Typically the measure is Bill of Materials (BOM) cost; as manufacturing techniques improve and Moore's law[5]

[4] In [Lindholm *et al.* 2003, p. 172], 'smartphone' is still an emerging category defined by its balance of phone, personal productivity, imaging and gaming features.

[5] Moore's Law, derived from an article by Gordon Moore of Intel, originally published in Electronics Magazine in 1965, is popularly formulated as predicting a doubling of computing power every 18 months.

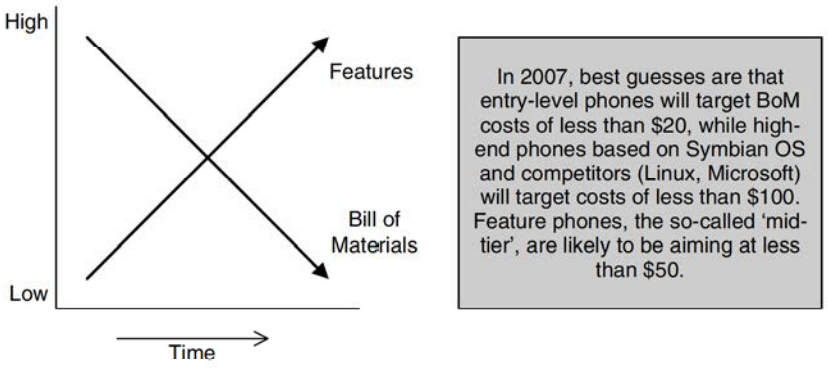


Figure 15.2 BOM costs fall but feature pressure rises

continues to hold, BOM costs fall (See Figure 15.2). As volumes continue to push inexorably upwards, driving marginal costs down, so the cost of a given feature set drops inexorably.

Meanwhile feature pressure (the demand to pack more and more functionality into phones) exerts a degree of counter pressure, driving ROM/RAM peripheral hardware requirements up. (For example, the camera-phone has evolved into the multi-camera-phone; still pictures have become video sequences; the ringtone phone has become a music-playing phone and has subsequently evolved into a direct competitor to MP3 players.) It's not clear whether that means that the BOM costs are not falling as fast as they might and therefore the line between high-end and mid-range is not falling as quickly as it might or whether it just means that we are all migrating to the high end.

Supporting the User Model

Phones are branded goods, fashion items, consumer appliances (see [Lindholm *et al.* 2003, Introduction]), and a host of other unlikely things that drive user expectations for how they behave, how easy they are to use and what they do. Phones have hard keys and soft keys; some have keyboards, some have pens, some accept voice commands; all may raise issues about handedness and screen orientation. Users expect 'natural' interaction models, associate interaction styles with brands, and fuel both performance pressure and feature pressure (where the performance and features of mid-range phones are increased to match high-end phones).

Supporting the Market

Phone manufacturers typically want rapid product cycles as part of their drive towards increased volume of sales. Feature pressure (again with a helping hand from Moore's law and the economics of volume) drives

a rapid technology-proliferation cycle. The pace of the drive towards increased volume of sales continually quickens and the cycle times for technology moving from research labs to products speed up.

The result is a demand for continuously greater agility from suppliers, including the operating system supplier, as well as continuously greater predictability, fuelled by the relatively long product lead time coupled with a short product cycle and lifetimes.

There is additional pressure on a system when it becomes a platform (being a supplier is easier: there is only one customer to please).

1997: The State of the Art

It is worth recalling what a mobile phone was in 1997, the year the Psion Series 5 was launched.

Mobile phones had been mass-market products since perhaps 1994–5, depending on which geographical area you look at, with the UK's penetration of a little over 20% about average for Europe, excluding Scandinavia (which was more than twice that). Penetration in the USA was a little ahead of the UK, Canada a little behind, and in the rest of the Americas almost non-existent, as it was in China [Haikio 2002, pp. 157–9].

Worldwide, mobile phone sales were just under 108 million units (in 2005, just under 800 million units were sold).[6] Motorola was dominant with 23% market share, Nokia was a little behind with 19%, and Ericsson was a little more behind with just under 15%. Nokia, indeed, seemed to have stumbled, issuing profit warnings in both 1995 and 1996. Otherwise, Vodafone in the UK had a little over two million customers (15.5 million in 2005) and had just introduced a pay-as-you-go service. The WAP Forum had just been created, with the first WAP phones two years or so away.

Motorola's phone of the year was the SlimLite (see Figure 15.3a), with a 4×16 character monochrome display and a 100-entry phonebook memory. Among Nokia's hot phones was the 3110 (see Figure 15.3b), in which a new, easy-to-use 'one key' (the Navi-key) user interface debuted. The design context was still dominated by users' propensity to use anything that looked like a dedicated 'Call' key to try to get a dialing tone before keying in a number. One of Navi-key's goals was to help

[6] The statistics and product specifications in this and the following paragraphs come from public sources. Some useful URLs include

- ***www.gartner.com/press_releases/asset_132473_11.html***
- ***www.gartner.com/5_about/press_room/pr19990208a.html***
- ***http://en.wikipedia.org/wiki/List_of_mobile_network_operators***
- ***www.gsmarena.com/motorola_slimlite-78.php***
- ***http://en.wikipedia.org/Nokia_7610***
- ***www.paconsulting.com/news/by_pa/1997/by_pa_19970115.htm***



Figure 15.3 a. Motorola SlimLite and b. Nokia 3110 phones

users learn to 'punch in the number before trying to contact the network' [Lindholm *et al.* 2003, p. 75].

Typical screens that year were 84×48 pixels, monochrome, giving 3 + 2 character lines (3 lines of 'user' text and 2 status lines). A typical 1997 Nokia (based on their DCT3 hardware platform) had 400 physical parts ([Lindholm *et al.* 2003]). DCT4 arrived in 2001 and halved the number of parts to 200.

Mobile network data services were largely unused except for SMS, which the analysts were still describing as underused. But the two-box PDA–phone solution, using a GSM phone as an infrared or serial cable modem, was seen as the coming thing for enabling email and Internet access and fax transmission from PDAs via phones.

The still somewhat revolutionary alternative could be glimpsed, though, in the Nokia 9000 (Figure 15.4), Nokia's first generation Communicator and the first converged PDA–phone device, introduced in 1996.



Figure 15.4 Nokia 9000

Convergence

Convergence may or may not have been inevitable but, despite its clunky physical form factor and monochrome display (640×200 pixels), the Nokia 9000 had put it on the cards. (It was a big, exciting, and very secret project inside Nokia; so much so, according to [Lindholm *et al.* 2003, p. 74], that no-one wanted to work on basic phones, which suddenly looked 'trivial' in comparison. Lindholm's Navi-key project had a hard time competing with it for resources.)[7]

The two-box solution, pairing a data-centric handheld such as the Psion Series 5 with a communications-centric GSM mobile, still seemed to be where the market was leading. Convergence, on the other hand, seemed to suggest putting a phone into a Series 5 or, more to the point, putting the Series 5 operating system into a phone. At some point, it became the inevitable next step.

Putting a Phone into the Series 5

The challenge which Symbian's engineers faced was to take an operating system that was designed and created with an almost obsessive attention to the details of a particular product context and evolve it to suit a different product just as well. It does not seem far to travel, after all, from the (ARM-based, pocket-sized, 1/2 VGA screen, clamshell-case, keyboard-centric, battery-powered) Psion Series 5 PDA to the (ARM-based, pocket-sized, sub-1/2 VGA screen, clamshell-case, keyboard-centric, battery-powered) Nokia 9210 Communicator. And it seems not that much further from the Communicator to an even more phone-like and less PDA-like device, in fact to a mainstream (if high-end) phone.

The Symbian OS mantra 'built for mobile phones from the ground up' doesn't quite tell the complete technical story. The operating system which shipped in the summer of 1997 in the Psion Series 5 knew exactly what device it was built for: a clamshell, AA-battery-powered, always-on PDA, with a keyboard, a touch screen and a couple of serial ports. It was optimized for mobile devices, but it also required several evolutionary steps to properly address mobile phones specifically.

Peter Jackson:

I don't know the point at which we really got to grips with the idea that we were making an operating system for a phone. Because when we started we weren't, we didn't. We were making an operating system for a Series 5, not even for a generic PDA but for a Series 5, and for other things like it that we might invent.

[7] Lindholm was later the architect of the Series 60 user interface and arguably the driver behind its platformization.

15.4 Telephony

Telephony services in Symbian OS are organized around the ETel server and framework, which is at the heart of the application-side interaction with the phone baseband or, as it was originally conceived, any modem at all.

As Andy Cloke remembers it, development work on ETel started even before the Series 5 had shipped. See Figure 15.5.

> **Andy Cloke:**
>
> We started doing ETel when Roger Nolan[8] was running the Comms group. I don't think it was entirely clear that we were going to do the next Nokia Communicator. At least, it wasn't clear to me at that point. There was still quite a focus on PDAs. Certainly, the thought that we wouldn't be doing PDAs hadn't become clear. So it could have been PDAs in a variety of forms: the modem could have been built in; it could have been a plug on; it could have been wirelessly linked. It was quite different from the way we are today.

The device that came to market was the Nokia 9210 Communicator, the direct descendant of the Nokia 9000. But at least two other phone-based projects ran more or less concurrently with the Communicator

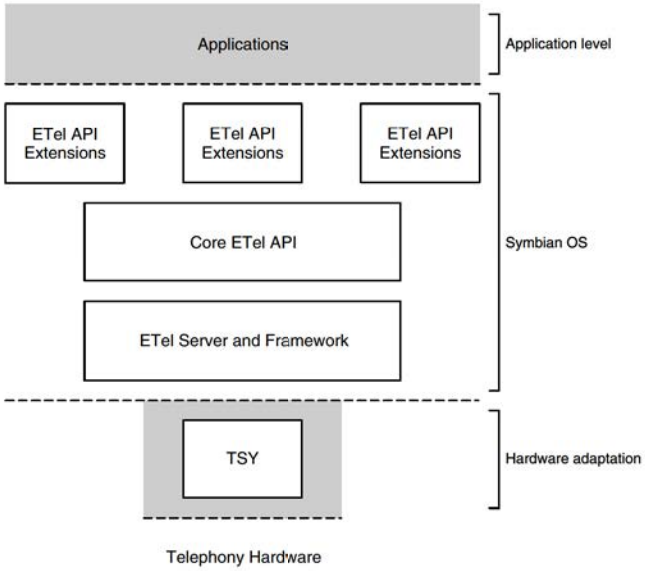


Figure 15.5 Telephony architecture

[8] Roger Nolan had previously been a member of Colly Myers's kernel team.

project, one for a Philips phone 'companion', which did not come to market, and one for the Ericsson R380. (See Chapter 2 for more about the background to these early projects.)

The starting point for the design was not really a converged device at all. While assumptions had moved beyond being simply an operating system for the Psion Series 5, the thinking was closer to a 'super PDA'.

Andy Cloke:

We were thinking about the possibility that you might have multiple modems that may appear and disappear. So the concept of PCMCIA cards that could be inserted and removed dynamically was still very prevalent. So that was the motivation for having multiple phones: you might have an internal modem and a PCMCIA modem and another one accessible over Bluetooth, for example.

ETel first appeared in the codeline at version 001 in July 1997 and by the end of the year was providing basic 'Hayes' control of a GSM phone (as modem) over a serial connection. As it happens, the test phone was none other than a Nokia 9000. If there had been uncertainty about whether the Nokia Communicator project was the target, it was resolved by the end of the year and ETel was explicitly delivering into the Nokia Communicator project.

Andy Cloke:

We established the ETel API first and then, in parallel as I remember, we started developing an AT-command-based TSY, and we didn't quite realize how big a job that was. And in parallel with doing that, we started talking to Nokia, to the Communicator development team. I remember going to talk to them and we thought we were going to get completely toasted when we presented our ideas because we were fairly new to it, and clearly these people knew everything about telephony and we didn't. I can remember coming back on the plane after the meeting in Tampere and thinking 'Wow!' Because we had expected to get roasted alive and that hadn't happened, so we just thought, 'How great we are, we've managed to get it right first time!' It wasn't until later that we realized how easy they'd been on us and how much we had to learn. In the end, we got strong design steers from Nokia and Ericsson.

In 1998, the first GSM extensions began to appear; the fax server was integrated from a standalone component into the ETel framework (as a framework extension). By the end of 1998, the code had reached a degree of stability as an alpha-release component of what was by then the 'EPOC Release 5' platform, aimed at the new Psion Series 5MX, a souped

up palmtop intended to have full phone and networking capability (in two-box mode).

In January 1999, ETel branched for the Ericsson R380 and by May it had become a component of the ER5u baseline, the so-called 'wide' or Unicode build of the operating system, and part of the main codeline. By then, there were multiple licensees taking the component.

Andy Cloke:

As you can imagine with companies working together but competing in the same market, there were quite a few political considerations; the opportunities for getting the different parties into a room together were fairly small, and the amount that they would discuss in the room together was also fairly small. You have to understand that we hadn't shipped a phone at this point, the Ericsson R380 hadn't shipped and the Nokia Communicator (9210) hadn't shipped either. So it was difficult for us. In terms of the TSY development, both Ericsson and Nokia were saying, 'Don't worry about that, we'll do the TSY'. We all thought that this was a good thing, little knowing that with us creating the API and them creating the TSY that plugs into the bottom of it, we were creating an integration nightmare. If we'd done the TSY design more in parallel it would have been better. We would have avoided some pain later.

ETel Design Goals and Architecture

ETel's design drivers are clear enough. It was designed to support multiple clients and multiple dynamically loaded TSY modules with the goal of enabling a PDA to access either a built-in or external modem – indeed, multiple modems at any time – typically to enable data calls (for example, SMS messaging and fax) and Internet access (for example, email and Web browsing). From the beginning, there were also some clever phone-specific extras, designed for the case in which the 'modem' being accessed was in fact a GSM mobile phone. For example, SIM toolkit functions allowed synchronization of on-phone SIM-stored and memory-stored phonebook entries with the PDA contacts application.

The design took the existing serial communications architecture as its starting point (Figure 15.6). The design goal was to provide an abstract model for controlling phones from Symbian OS. Phone hardware was understood in classical Symbian terms as a resource to be shared by multiple applications with serialized access; in other words, the server model applied. Analogously with CSY serial communications modules, TSY telephony modules were defined as the abstractions for actual hardware and a similar framework architecture to the serial communications system was adopted. Depending on what hardware was available to the system (and what application was requested) the TSY would either interface directly to the hardware (the baseband–built-in modem case)

or access the hardware through the serial communications system (for example, a TSY sending AT commands to a true Hayes modem or to a GSM phone presenting an AT interface, over infrared or a cable serial connection).

The ETel architecture closely followed the tried and tested architecture of the Comms services. A classic Symbian OS client–server interface shares access to telephony services and hardware between multiple clients. A framework architecture provides for a core API which is extensible in two directions by plug-ins; horizontally, ETel extensions add richer functionality to the basic core set with extensions for fax, packet data for GPRS and 3G, the Multimode extensions for CDMA2000, and SIM toolkit extensions; vertically, plug-in TSY adapter modules, modeled on the Comms CSY modules, that are loaded on demand interface the abstracted ETel APIs to the actual hardware available.

ETel therefore interacted closely with the serial communications system and, in fact, did so through the generic mechanism of the Socket Server (requesting a serial socket connection, for example). Looked at from the other direction, ETel equally became a socket provider, providing telephony sockets to serial or networking components.

The communications design analogy was an obvious starting point, especially because the design was proven and had shown itself to be both extensible and flexible. In effect, the whole communications architecture was elaborated horizontally so that the telephony system was created as

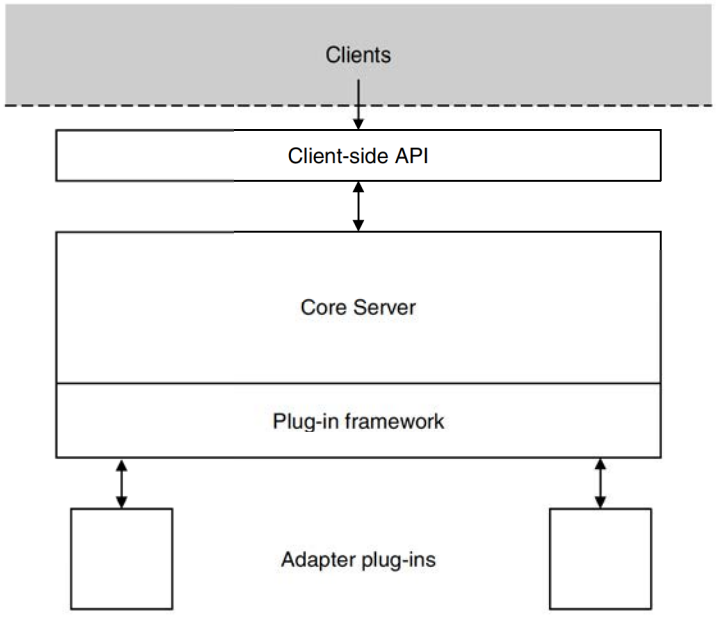


Figure 15.6 ETel design mirrored the design of the serial comms server

a peer of the networking and serial communications systems. The serial system was 'first among equals', primarily because, in the use cases that drove the design, serial communications via modem always provided the physical access to the network (phone or Ethernet).

With hindsight, however, the PDA or two-box use case dominated over the built-in phone use case. ETel started with a design goal which it met admirably, but which was rapidly overtaken by the change in context. Symbian OS in a phone has different telephony requirements to Symbian OS in a modem-connected PDA.

Andy Cloke:

Our early design assumptions were not really true any more; modem wasn't the primary use case. And certainly the primary use case that we're dealing with now is a phone that has a single baseband, and consequently ETel has morphed into a baseband abstraction.

The more complex question is whether ETel's design supports the needs of an abstracted baseband or simply provides an API for application access to the baseband.

In design terms, ETel is a classically good example of object-oriented abstraction. Its three key concepts are phones, lines and calls. Clients request an `RPhone` session, from within which an `RLine` subsession can be opened and a further `RCall` subsession can be created.

The TSY instantiates corresponding derived classes from the framework: `CPhoneBase`, `CLineBase` and `CCallBase`. In turn, these objects create AT commands, instantiated as objects derived from `CATBase`, which are then sequenced through the TSY's command sequencer, which controls a communications port requested through a serial communications session owned by the TSY. The communications port in question provides a direct (docking-style), infrared or serial-cable connection to an actual phone, which responds to the AT commands.

Andy Cloke:

I wish in hindsight I had spent more time looking at the ISDN specs, standard wireline phone specs, and the GSM standards. I think `Phone` and `Call` are still quite valid abstractions, but I think `Line` was a bit of a waste of our time. It was there originally because, on a single GSM phone in the GSM specs, you can have a different telephone number for voice calls, data calls and fax calls.

Fax, indeed, turned out to be troublesome.

Andy Cloke:

Fax over GSM is tricky because of timing issues. You have to spoof packets on the base-station side to stop the fax machine timing out because the transmission delays are too long over GSM. Well, and then it all gets more complex because of the alternate numbers. But anyway, I think I would have dropped the `Line` abstraction.

Another, subtle, design assumption that proved a problem comes back again to assumptions about the nature of the modem that is providing access to the phone network.

Andy Cloke:

That's something else that ETel was predicated on, that the modem would become completely commoditized, and while it is slowly moving in that direction, with the advent of 3G and all the different services on there and all the different ways that packet data works and some of the services work, that is not true yet. And coming you have HSDPA and HSUPA – High-Speed Downlink and High-Speed Uplink Packet Access – which will be the next service that will be required. It's already being referred to as 3.5G. So these sorts of forces stop the modem software – I'm using these words, modem and baseband, interchangeably – they stop that software becoming a commodity because it just moves on and on, it's moving so fast. How many non-proprietary 3G signaling stacks are there out there that have any kind of market credibility? I think the answer is as small as three or four, ones where you're going to buy off the shelf, so it's certainly not yet a commodity item.

The Problem With ETel

Charles Davies:

We did a telephony API, whereas what we needed was a distributed computing solution with a baseband. Or to put it another way, it was not so much a telephony API we needed as a subsystem that does more than telephony, but of which telephony is one application.

The more general case of supporting data communications, including abstracting the baseband, is the direction in which Charles Davies is heading. With hindsight, the requirement was not to provide abstract

control for a phone, because outside the two-box context that is not what devices require. What devices really require are high speed data connections between the application side and the phone baseband. As Davies puts it, something much more like what he calls 'a distributed computing solution with a baseband.'

Charles Davies:

What that really means is that you supply a reliable by-value RPC mechanism between the two sides which is independent of the application you are doing it from, and then telephony would be one of the applications.

On the question of the importance of telephony, Charles Davies is even more radical.

Charles Davies:

You can debate how important a telephony API really is anyway. Many other APIs are much more important from the point of view of the operating system, in the sense that connections to packet data are more important, because telephony is only needed by one application, although it certainly is an important application. But it's all the rest of it that allows a phone to be built.

Andy Cloke takes a more sanguine line.

Andy Cloke:

With hindsight we would have preferred to establish a narrower API upon which Symbian would have built its services, in other words, data calls, establishment of PDP contexts and secondary contexts, transmission of SMSs, reception of SMSs, these sorts of things. It's very important that we have an API which works well there, a downward, hardware-adaptation-interface-style API.

The Danger of the Thin API

For Andy Cloke, the real lesson here is a more general one about API design in the case where both the application-level API clients and the hardware-level adapters are extrinsic to the operating system, and the operating system has the role of defining a thin, stable API against which partners can develop their value-adding components. In this case, two big issues arise: how thin can a thin API be and still be sustainable? And,

how do you manage the design and development process to ensure that the API is viable and the right one?

Andy Cloke:

ETel today is a baseband abstraction layer and suffers because it tries to provide an abstraction for the whole of the baseband. Not just modem functionality, but also the ability to make calls, put calls on hold, do multiparty calls and all that stuff, and it also contains all the supplementary services, call forwarding, notifications that somebody else has put you on hold, so all of this kind of Layer Three GSM signaling, all these notifications need to come through. As well as that, it also has to be able to transmit SMS messages, which are actually a reasonably complex beast when you dig down because there are various different levels of ACK-ing that occur in the SMS protocol stack, and then there is USSD too, the unstructured service data.

Symbian does not build the telephony application above ETel, and we do not own the TSY below ETel. Where you have code like that which is so thin between two parties, between the top-side API users and the bottom-side API users, especially when they're very broad, well it's almost just worth getting out of the way and letting the users sort it out between them. Actually an API produced by the creator of a telephony engine might well deliver the best result.

It is possible, of course, to define, create and successfully manage 'thin' frameworks. There are successful examples in Symbian OS, as well as industry examples.

Andy Cloke:

A good example is Direct X, although it's in a slightly different area. You have the games developers on one side and the graphics drivers creators on the other side. It's a tricky path to walk, creating that kind of thing. You have to have everybody bought into the fact that you need it, and then create a forum with a number of graphics card creators and a number of games developers, and you need to mediate between them, which can be quite hard. Similarly, this is a hard thing to do in the telephony area.

What makes it difficult in the case of ETel is the very specific dynamic between Symbian, as platform company, and its licensees, as phone manufacturers. Because licensees already have their own solutions in this area – it is, after all, their very particular expertise – they do not necessarily see this as a place in which the operating system either can or should try to add value. From the Symbian perspective, however, giving up its telephony offering would reduce the value of the platform for potential new licensees lacking existing investments in telephony. Symbian stands in the centre of these conflicting positions.

Andy Cloke:

Some licensees do not see this as something they require. They would like to talk straight to the silicon, direct to the baseband. They don't want anything in the way. So they just want to know what is the minimal TSY that they need to create in order to support Symbian OS on top of it, and the rest of it they bypass. They just talk straight to the baseband.

But that's not the position of all licensees. For example, it is a matter of public record that Nokia does not license its own telephony application (its crown jewels, or part of them) to competitors. It is important therefore for the viability of the S60 platform that its licensees should be able to integrate their own (or a third-party) telephony application. For these licensees, an application interface is critical and it is also critical that it is supplied as part of the operating system, so that it can be standardized and controlled, since without a Symbian OS API between the application above and the TSY below, there would be no standard for the interface between the two, or at any rate no controlled standard.

This implies a clash of interests between Symbian, those customers who are in the business of making complete telephony solutions themselves and simply want the rest of the operating-system functionality, and those who may not have a telephony application at all and want the operating system they buy to offer a complete solution.

This is, of course, not a design or engineering problem; it's a business problem and there is no engineering solution to it. What is the lesson?

Andy Cloke:

You should do an incremental design and you should also make sure that you have an interested community both above and below the interface and that you involve them, that you validate the interface by continuing effort on it through the development process of the clients and the plug-ins on both sides, really to prevent yourself being circumvented wholly by those people. Especially when you have the same company building both the client and the plug-in. Because they will build assumptions into their client code which are fulfilled by their plug-in code, but which are completely absent from the specifications. And of course they will extend it further than you would ever expect.

15.5 Messaging: It's Different on a Phone

Messaging is another area which proved tricky in the transition to being a full-on phone-focused operating system. The Psion Series 5 was

specifically intended to provide integrated email and, in particular, to be 'Internet ready', offering standard, Internet-based mail solutions, as well as access to other Internet services. Support for Internet-based email was therefore an important design driver.

Keith de Mendonca joined Psion in 1994 in one of the early, small expansions of the company which came with the success of the Series 3 and the start of the preparations for the Series 5 project.

Keith de Mendonca:

I started with SDK work as a grounding, and I was working with Colly Myers on the Psion remote comms protocol SDK for the Psion Series 3a. Interestingly enough, I remember getting a fax of appreciation from a small company in America that I didn't really know, a small company called Palm Computing that it turned out we were in communication with about them perhaps writing connectivity software.

At around that time there was talk of cash-rich Psion buying Palm. But in the end Psion and Palm went their own ways. Meanwhile, after a stint writing applications for some of the later Series 3 machines, de Mendonca moved to the new Messaging team.

Keith de Mendonca:

I think at the time it was the biggest team that Psion had ever had working on a single application.

Its first task had been to create a messaging application for the Series 3.

Keith de Mendonca:

On the Series 3 the focus had been on working just with corporate mail. It was very much before the real Internet explosion, so it was just bespoke. It worked with ccMail and I think one or two other mail clients. It was only later that there was a decision to go Internet, where the open standards were.

For the Series 5, the team started from scratch with a new, modular and flexible messaging-client application which was designed to provide a seamless, unified, single-point-of-access interface to email and other message types.

Keith de Mendonca:

The intention for the Series 5 was very much that open standards were the future for us and the company. This was the most flexible way of actually interacting with servers and hence getting the best market share. So all the work really was to do POP3 and SMTP and IMAP4, which we started doing for the first time. In addition to that it obviously did SMS and there was a fax component as well, so you could send and receive faxes. So that was all in the messaging application and the design of the application was informed by discussions that we had been having at the time with Nokia, because obviously in the background there had been lots of work thinking about what was to be in the Communicator.

The Nokia Communicator project was already having an influence on the design. The ambition level was high.

Keith de Mendonca:

That was probably representative of the Nokia Communicator requirements generally, that what everybody shipped in the end was much less than what we intended to do when the project was scoped at the very beginning. I think the idea about what this next generation of communicator should do was beautiful and visionary, but the vision was very much greater than the reality of what both companies could produce in the time available.

Messaging Design Goals and Architecture

The key design drivers were to support open standards, with a flexible and extensible solution. The core of the design was a message storage framework for all messages, regardless of type, with plug-in protocol modules for particular message types. This was a modular solution, based on sound object-oriented principles of abstraction from a generic notion of message to the individual message types, designed for openness and extensibility.

Keith de Mendonca:

You had one message store but you could actually plug in different modules that allowed you to add different messaging protocols as and when they came up and even add those after market. So you could literally just download some SIS file and install some components, and suddenly you had fax or IMAP4 support, for instance. So the message architecture was designed from the very beginning to be very modular and flexible and that is the architecture which we still have today.

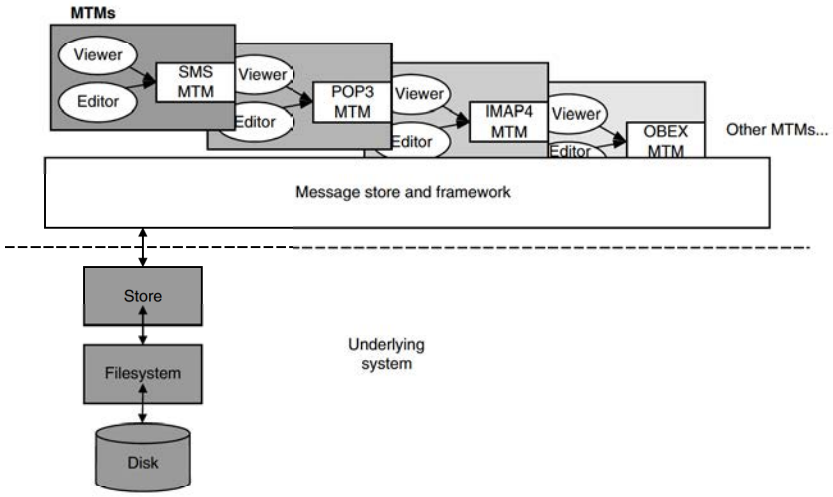


Figure 15.7 Messaging architecture

Following the general principle of Symbian OS design, access to the message store is through a server, which offers a client-side API. The plug-in modules (Message Type Modules or MTMs) are loaded dynamically by the framework on demand, based on the type of the message, and handle everything from the bitmap which is displayed in the inbox to indicate the message type and status, to creating, copying, moving, deleting, parsing and editing contents of the message type. See Figure 15.7.

Keith de Mendonca:

It also provided a back door for any special commands, which was literally just a special ID and any arguments that you chose to send to it in case those generic ones didn't actually represent what you were trying to do with a particular message type. The framework also allowed a simple API for another client of that server, another application, to send content as a message and choose the type dynamically based on the plug-ins which happened to be in the framework at that time, the Send-As API.

While the design met the stated requirements, the result was a large and complex component with complex APIs. In at least one area, the flexibility of the application had an immediate pay-off. With the messaging application still not ready to ship on the final Series 5 launch date, the messaging architecture made it possible to ship a complete messaging package as a user-installable after-market upgrade. It duly shipped some months later.

That flexibility proved just as useful for the first release of Nokia's 9210 Communicator. With the product pushing the limits of its ROM budget,

messaging saved the day by shipping some features not in ROM but as optional installable packages on a companion CD.

Keith de Mendonca:

When we ran out of ROM space on the Communicator they decided to put the fax MTMs on the CD instead of built into the actual machine. It allowed them some flexibility in the ROM. Likewise, IMAP4 was originally delivered as a SIS file because it was finished a little later than the messaging application, which kind of proved that the architecture worked, you really could deliver things after market.

The flexible architecture and, in particular, the support for after-market delivery of MTMs by third parties and partners, which enabled messaging capabilities to be extended for any future message types, had been among Nokia's headline design requirement.

Keith de Mendonca:

It was responding to a customer-level requirement as far as we were concerned. But I'm not really sure to what extent that flexibility was really needed and you could argue that it was over-engineered or the requirement was a bit too heavy-duty, for the actual reality of what the application required. There was a vision of a new community of programmers writing corporate email plug-ins and suchlike, but the downside of such an extremely flexible architecture is that, unfortunately, often it can be quite complicated. This was quite a barrier for programmers and there's not really been a large market for those kind of MTMs to this day. But of course then you are left with your architecture, which you need to maintain and continue. A lean development model might have delivered the best value first, and then grown it as and when the appetite of both companies grew and we understood better what the market really wanted.

Perhaps one of the more puzzling facts of the phone market to date is that, despite the apparently huge popularity of products such as the Blackberry, email on phones has not yet proved a core function. In general, messaging – other than SMS – seems to have made little impact on users to date.

Keith de Mendonca:

The Communicator was very advanced but also much of the same code went into the Ericsson R380, which was much smaller, but that was a really visionary product as well. Everybody in the messaging team has picked up their mail on their phones ever since that day whenever it was, 1999 or thereabouts, yet

that's only becoming a relatively recent phenomenon for other people buying phones. So it was extraordinary power that there was inside the actual product right from those earliest days.

What is the lesson? That making a phone operating system is not just about putting cool technology in the box. The harder formula seems to be – the right technology at the right time in the right product for the right market.

Meanwhile, flexibility implies complexity and the complexity of the messaging architecture has, arguably, had its downside.

Keith de Mendonca:

If I had my way again we would have presented less complex APIs for messaging. This complexity is one reason for the small number of people actually innovating using messaging. For instance, it's quite a big job to port your own corporate email solution that you might already have running on WinCE or some other device, to then package those into the MTMs, understand that paradigm and make it work on a Symbian phone. So it's been a barrier to that.

But also performance became an issue, primarily with email, due to some design decisions that were made early on. We also have been held back, perhaps, by feeling very constrained by APIs and data structure storage methods. That looks ridiculous looking back now. Of course we are now very closely restricted by such considerations, because we sell so many phones a month, but I felt very restricted about changing those things in Symbian OS v6.0 or between v6.0 and v6.1, when we would have survived making changes. But there was this compatibility promise very strongly enshrined in Psion and Symbian in those times, that you couldn't break binary compatibility, so it was quite difficult to make substantial changes.

The Universal Inbox

There are two other lessons from the messaging experience, one about designing for the wrong use case and one about performance.

Keith de Mendonca:

Although we knew our primary goal was to produce mobile phones and the phone would always be there, there was still quite a lot of work, say in the Symbian OS v6.1 project, for two-box support, even though the particular product never came to fruition. That probably resulted in the two-box concept

not dying so quickly, although the company was focusing 100% on its new environment.

One particular issue concerns the design of the message folders.

Keith de Mendonca:

When we designed messaging we had a concept of 'the universal inbox' [See Figure 15.8], so it didn't matter if it was SMS or emails or whatever, they would all appear in the same message store and appear physically in the same inbox view. That's exactly what we used to have on the Series 5, where, of course, it was a two-box solution. You put your phone next to it or connected a modem to it and you downloaded all your email, that was the concept, and it all appeared in the inbox. And if you synchronized your SMSs, they would appear there too.

But from the beginning we also started toying with an idea of offline operations, a remote mailbox view. The point was that because you were a small device and you didn't have that much memory, you didn't want to download everything, because you didn't know how big those things were going to be. Quite frankly you might not want to have them. So you would synchronize the headers, that's all that appeared in the inbox, and then if you were interested in anything, then you would populate the individual items and fill them up with their content. The view was then represented as an external view, those messages didn't appear as just headers inside the inbox, they were actually shown as a separate independent view representing to the user that they were looking somewhere else, and in fact they were inside my very nice tree-structured message store browsing the complete structure of the message contents, which was very expensive in performance, not at all like just looking at the headers.

Now, in effect, the view on a phone is actually that offline view, certainly it's the way that Nokia displays them, it's the view of your mailbox. You no longer just synchronize the headers, on a phone you go straight into browsing and opening and reading. So the concept of the universal inbox immediately dies, because in the new model that we followed you had a remote view of each of your email accounts and those messages never appeared in your Inbox.

The difference between browsing the simple, inbox view of the message headers and browsing the elaborate, tree-structured representation of the complete contents of messages has a significant performance impact, one which, moreover, is tricky to resolve because the design is being used in an unintended way.

There are other impacts too, which ripple down throughout the whole design. For example, the basic assumptions underlying the design of the messaging APIs are the wrong ones for some of the most common phone use cases.

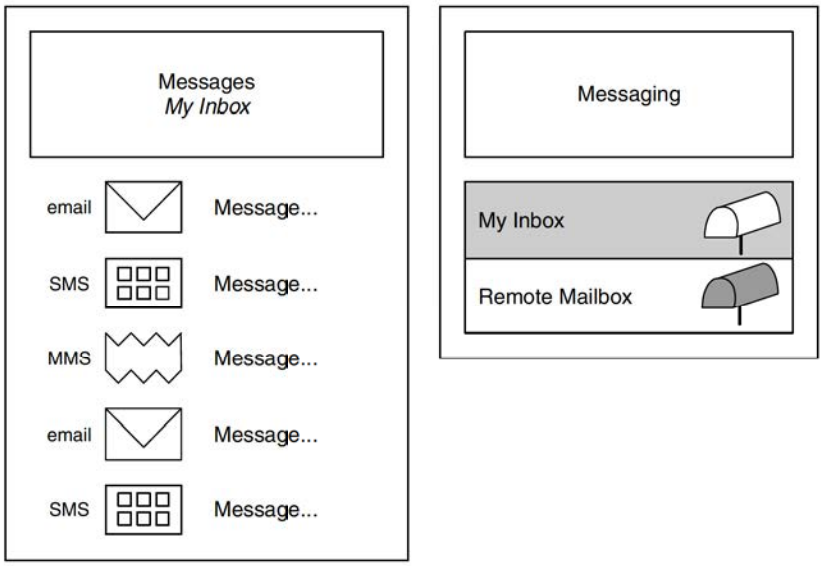


Figure 15.8 The universal Inbox

Keith de Mendonca:

Consider the generic API for the MTMs, which includes Copy and Move, for example. Actually, you never copy or move anything to your inbox on our phones. Those methods on that generic API for all of those MTMs are probably never used nowadays, because the API doesn't match the common usage, so the methods are redundant. Should we have predicted that? But take a different example, along comes MMS and suddenly our original assumptions about the universal inbox start to make sense again, because now you can have a mixture of SMS and MMS messages in the inbox. It's become universal again! But it's only when that MMS technology reared its head that you actually had something else to go in your universal inbox, because remember, emails are always set off alone in their own view anyway. But can we claim that our design was right after all?

It is not so much a point about user interface design, because the underlying conceptual design is interpreted by the user interface in a way that suits the user interface paradigm.

Keith de Mendonca:

Of course it's up to the UI how it actually presents those things. On UIQ phones, it doesn't mix them all together in the inbox. It shows you different message types and it 'pretends' they're all in the same general location.

Conceptually they're stored in a different structure underneath these remote services.

The underlying conceptual design does, however, have a strong impact on how the system APIs below it, and which support it, are used. In the case of the original design of messaging, the supporting APIs are misused.

Misusing Store

Keith de Mendonca:

The conceptual model for the message store is a logical tree structure with the root at the top. Underneath the root there are the folders that you know about, the Inbox, Outbox, Drafts and Sent, also some representations of other things that are on the device; those are all seen as local services. It also has the concept of remote services, so your email at work technically would be the SMTP email server at work, or there's the fax machine you are trying to send to or the SMS service centre that you're sending your message to. Normally those things are invisible, but they still exist physically and are represented inside the message store.

Symbian OS supports folders, and it's very flexible – you can create any number of folders and put any number of messages in folders. Logically that's just a tree stored in RAM, in a RAM index effectively, but also it's written onto disk in case the battery fails, and it uses the Stream interfaces which are part of the Store in Symbian OS. It was designed in the same way that we designed most things, to be robust and never to lose data if you lost power. Therefore there's a lot of writing to disk, and as we know and have learnt, the Store components themselves, because they are so careful in making sure that you can never lose any data, are also quite expensive in disk writing.

This wasn't a problem when we wrote the message architecture, because it was designed for the Psion Series 5 which had a RAM disk and didn't have any performance problems. However, it started to create some problems if you put your message store on a removable CF card, which was the first-generation flash technology.

The team made minor modifications to their implementation when performance problems began to manifest themselves on real hardware. The fact that development was in the first instance carried out under emulation didn't help matters and the inevitable desire of licensees to keep prototype hardware under wraps didn't help either.

Keith de Mendonca:

A lot of last minute changes were made to try to improve the performance, including for the Symbian OS v6.0 release which went into the Communicator. One change was that we decided we couldn't afford to use Store for those small files, and we actually created our own store without changing the API. So from the public interface it would seem as if Store classes still exist, but we just wrote a binary file format which was faster, with some loss of robustness if the battery failed.

Also, email messages were described in the filing system not as a single message containing all the data for an email, but as a tree of objects representing exactly the MIME structure of that email. This meant you had an awful lot of files or objects for representing one message. This was true especially for HTML-style emails which are the standard way of sending emails now, because they not only send you the HTML version which has the complete content associated with it, but they also send you a simple text version as well, and of course we stored that as well. You may have 8 or 10 individual files representing an email, and if you have 8 to 10 files to write then obviously it will be slow. That's a basic challenge that we have even today, the performance of downloading emails, where we still have to create all those physical objects.

As in any other areas of the operating system, performance issues are continually reviewed and implementations tuned. But no one likes to have been wrong-footed by designing for a use case which changes because the overall device design point changes, as happens, for example, in the move from PDA to phone assumptions.

Keith de Mendonca:

Obviously, you would like to start again knowing where a lot of the faults are. But we are still constrained, we can't break this binary compatibility which we have offered so long, because the cost to customers and others of changing the message applications is quite expensive if we make any changes. So we work within the boundaries of our compatibility requirements and we're always making improvements.

Another compounding problem was the unfortunate interaction of the Store design with the behavior of Flash memory systems. Having been designed in the context of a conventional, fast, random-access, RAM-disk-based file system, the design turned out to be weak in the context of sequential-access, fast-reading but slow-writing flash hardware. Store, designed to offer a transaction-style, robust, append-based file system

does not have a natural concept of updating already written data. Streams are appended sequentially to the physical medium and the header index is updated to point to them.

In the worst possible use case for a flash-based system, the design of Store ensures that it only ever grows and, in the worst possible use case, it even grows to delete things, depending for its space performance (so that it does not outgrow available memory resources) on a regular compaction cycle. The reason, of course, is data integrity, and that most basic, fundamental guiding principle of the operating system: 'Thou shalt not lose the user's data'.

To some extent, the problems with the early version of the messaging framework highlight a more subtle problem.

Martin Tasker:

The ground rules that we gave ourselves when we were making the PDA are not the same as the ground rules in the mobile phone space. We told ourselves that the ground rules are that you should assume you've got very little memory, and you should assume that the process is going to run forever, but also to assume that you're not in an embedded OS context where you can give a fixed partition to everything.

The thing is that in the mobile phone world, you do have system restarts and memory management is more complicated. If you look at the phone from a high-level perspective, don't just ask about the details of programming. Instead, ask about the dynamics of the number of applications and the number of services and how often it is turned on and off and how you allocate memory when you're sharing it between the programs, and look at the data storage and the RAM. The answers to those questions are different from those that we originally looked at in the PDA context.

It is not that the problem is one of greater complexity; it is different complexity.

For Ian Hutton, the distance between the early phone projects such as the Nokia 9210 and the Ericsson R380 and the phone projects that Symbian is involved in today, is the best demonstration that the hard lessons have been absorbed and the big step has already been made.

Ian Hutton:

I think the fact that we really are now beginning to see very, very ordinary phones, not just the incredible phones but ordinary phones with Symbian OS inside, phones that will still do anything you want, but are very similar to ordinary mid-range phones. In a way, that's the biggest step.

16

One Size Does Not Fit All: The Radical User Interface Solution

16.1 Introduction

Symbian OS is an inherently GUI-centric operating system which ships without a GUI of its own.[1] Moreover it targets a market (wireless devices generally and mobile phones in particular) in which the user interface is a critical competitive element. Delivering an operating system without a user interface is a radical software solution to a business dilemma: how to create a common, multi-vendor platform for phones, while at the same time not merely supporting but positively driving vendor differentiation.

This case study traces the evolution of Symbian's user interface architecture and strategy. User interface issues are particularly interesting because they highlight many of the unique problems of the mobile phone market. It is significant that Symbian's user interface architecture and user interface strategy have both undergone some quite radical transformations since the company was first created.

Although in computing terms the specification of a typical phone based on Symbian OS in mid-2006 is equivalent to that of a mid-range PC of the mid-1990s and exceeds that of a mid-range PC of the 1980s by an order of magnitude,[2] nonetheless a mobile phone is only incidentally a computing device. It is simply not thought of by users as being 'a computer' (What

[1] I use the terms GUI and UI almost interchangeably in this chapter; and use 'GUI' when I want to emphasize the graphical aspect in particular.

[2] 80386-based PCs clocked at a maximum of 33 MHz in 1985; 80486-based PCs reached 100 MHz by 1994; Nokia N series phones have reached 300 MHz in mid-2006. IBM's top of the range PS/2 machine, the 1987 Model 80 (with a list price of more than $10 000 – without adjusting for inflation!) featured a 16 MHz processor, 16 MB RAM, and 140 MB hard drive. The Nokia 5500 shipped with 64 MB RAM and supported removeable cards up to 1 GB.

is it then? It's a phone, stupid!). As I have argued in previous chapters phones are 'just different' from PCs and PDAs.

Phones have evolved rapidly from the clunky business tool of the 1980s to the seemingly essential 'upscale accessory' of 2006. In particular they have become consumer goods – 'and we cannot expect the user to configure them'.[3] But high-end phones have also become application platforms and this, in particular, has been the ground staked out by Symbian, which has deliberately set out to create an open platform for third-party software development. Where mobile phones began as the 'functionally direct replacements of their wired forebears', they have evolved a long way beyond those beginnings, boosted by the rich variety of available applications and by dizzying competition between vendors to outdo each other with new software and hardware features. They have become quite independent platforms for personal communication in modes both new (picture messaging, text messaging, and phoning home from the train, plane, street or shop), nearly new (email on the go) and old (plain old voice from home or office); for personal broadcasting (mobile blogging, web-sharable photo albums); and personal entertainment (games, MP3 players, pocket web browsers, pocket TVs). If they ever were just functional replacements for fixed-line phones, they certainly are no longer. With astonishing speed, mobile phones have become 'platforms for entertainment and commerce and tools for information management and media consumption'; everything in other words from business tools to games players to shopping tools to fashion accessories.

Differentiation: The Big Idea

The goal of differentiation is to avoid selling on price alone. Operators look for ways to apply their own branding to phones and to add value that will not be available from competing operators. On occasion they strike exclusive licensing deals or negotiate periods of exclusivity with vendors for particular phones. At the very least, most operators demand a minimum degree of customization of phones from vendors, for example operator-specific packaging, stenciling of the operator name or branded logo onto phones, and inclusion of custom operator applications or support for dedicated operator services on phones (for example, O2's Homezone and Vodafone live!).

This is not quite the same as vendor differentiation, which is the most visible form of differentiation. For phone vendors, differentiation – of their phones from those of their competitors – encompasses everything from design philosophy and style through reliability and build quality to ease of use and, of course, technologies and features.

[3] The quotes in this section are from Keinonen [Lindholm *et al*. 2003, p. 4] and Kiljander and Jarnstrom [Lindholm *et al*. 2003, p. 15].

Naturally all players in the phone market are seeking competitive advantage, but there is something else at work: avoiding commoditization. For both operators and phone vendors, commoditization is an ever-present threat and is one part of the complex dynamic which drives the exponential pace of technological advance and feature growth. 'Commoditization' means the cheap and ubiquitous reproduction of what was once expensive and unique. Commoditization reduces margins because it reduces competitive advantage to price competition. Commoditization naturally centers on hard technologies and features – 'do more, go faster'. Softer product qualities – 'do better, be easier' – are therefore critical points of defense against commoditization.

In the battle to maintain differentiation, the user interface has become a key competitive element for phone vendors. Indeed for phone vendors, design philosophy and style ('cool' is an important selling point for phones) and properties such as usability have become almost as important as features. Since these are all properties which touch or are touched by the user interface, the phone UI has turned out to be a key business competitive edge that drives market segmentation and, as a result, brand leadership.

As Christian Lindholm, the creator of the original S60 user interface puts it, the UI 'is one of the key elements in the fight for customers', creating pull from both end-users and networks, an essential 'competitive asset in the race for market dominance.'[4]

What's in a User Interface?

The user interface is where the phone software and the specific hardware features of a particular phone meet to enable the user to access the features of the device. In large part, 'usability' reduces to questions about the user interface and the myriad decisions made by its designers and implementers about such things as one-handed versus two-handed use, pen versus keyboard input, feature richness versus interface simplicity, file-centered versus task-oriented application design, and so on. At one level of detail down, these become decisions about such details as screen color schemes and fonts, menu structure and sequencing, how hard and soft keys interact with on-screen items, and so on.

At first glance, user-interface style may seem to be a slender thread to hang market aspirations on and a tenuous driver for market share or even dominance, but of course it is not the only driver. But the critical contribution it makes has to some extent become a reality for all consumer products (think of the iPod without its click-wheel), and it has certainly become a reality for phones.

[4] Kiljander and Jarnstrom [Lindholm *et al.* 2003, p. 15].

The Multiple GUI Operating System

To succeed in this complex context, Symbian's solution is to support multiple UIs and to engage with others to create those UIs. Licensees either buy in the platform pre-integrated and tested from a Symbian OS UI vendor such as S60 or UIQ Technology or they buy in the operating system 'headless' and develop their own GUI.

In architecture terms (see Figure 16.1), the topmost layer has been removed from Symbian OS. The operating system does not provide the UI layer; instead it provides the infrastructure (frameworks and primitives) for UI creation. Beneath the UI, the common frameworks and the underlying services of the operating system itself ensure substantial platform compatibility at the application-engine (i.e. application-logic) level. Indeed applications can be targeted at the different available user interfaces by customizing the application UI without changing the application engine. A Symbian licensee either creates a bespoke user interface from the frameworks (although this is not a trivial engineering effort, it is the option chosen by DoCoMo in Japan for its FOMA platform, for example) or buys in a UI from a specialist vendor (both UIQ Technology AB, which was spun out of Symbian as a separate company, and Nokia license UIs and supply a pre-integrated platform to phone vendors).

Symbian OS on a Sony Ericsson phone or a Motorola phone with the UIQ GUI – the P910 or P990, say, or the Motorola A1000 – looks and feels very different to Symbian OS on a Fujitsu or Mitsubishi FOMA phone or to Symbian OS on a Nokia, Samsung or LG phone with an S60 GUI. And yet the underlying operating system is the same and the very same applications can run on all of these different phones, sharing identical source code at the application-logic and data-model levels. While Symbian OS is tightly integrated to the GUI which runs on top of it, by way of the UI and application frameworks, it is designed to be GUI-neutral. For application developers and from the application

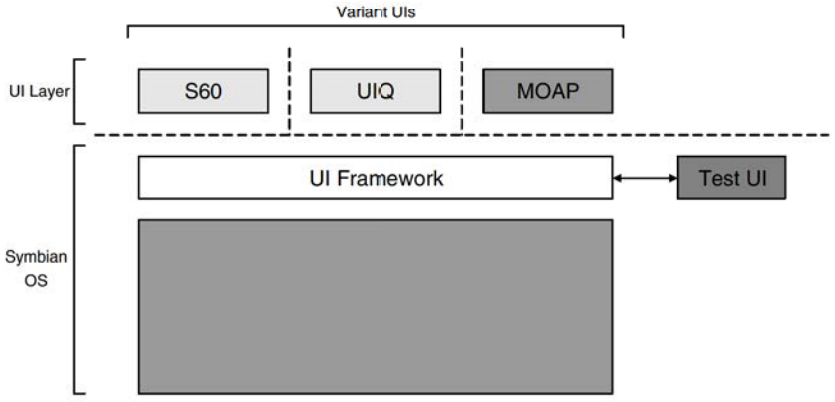


Figure 16.1 Symbian OS user interface architecture

perspective, although Symbian OS applications are intrinsically GUI in nature, applications are only loosely coupled to a particular GUI variant, since the application model and the application-event loop are enshrined in the operating system itself and in the UI Framework support, and not in the variant user interface.

Symbian's solution contrasts with that of other operating systems targeting mobile phones. Windows Mobile, a Windows CE derivative, is architecturally a monolithic UI, like its desktop parent. All phones that use it, from whichever phone vendor, share the common Windows interface and its Microsoft signature branding. Whatever the other opportunities for vendor differentiation, phone vendors supplying Windows Mobile phones are, to all intents and purposes, Microsoft OEMs. Windows Mobile is certainly a platform operating system, but what has so far made it unattractive to many phone vendors is the fact that it is Microsoft's platform.

The situation with Linux is somewhat different. To date, the Linux phones which have shipped (in large numbers in Japan and China) are not Linux platforms because they are not natively programmable. Instead, they are closed phones exposing only limited Java APIs. Further undermining the platform potential of Linux phones is the proprietary nature of the Linux distributions on which the phones are based, with the phone vendor typically directly owning the distribution. To some extent new 'open' user interface toolkits supplied by independent vendors for Linux-based phones (such as the Qtopia QT user interface toolkit from Trolltech) may open platform potential to Linux phones if they become widely adopted. However, it is something of an irony that the question of openness still remains a significant challenge for Linux on phones. Qualcomm's Brew platform, which so far has been limited to CDMA markets but which is now migrating to GSM, offers device vendors a choice between creating their own bespoke UI, or adopting Qualcomm's customizable uiOne user interface.

The early decision that Symbian made was that the phone market would reward UI diversity based on a common underlying platform. Thus far, certainly, the market has found Microsoft-style homogenization resistible.[5] Linux, fast emerging as possibly the strongest challenger to Symbian OS, has not yet solved the platform challenge. So far Symbian OS remains the only operating system which is open to third-party developers across multiple phone vendors, across multiple operators, and in all geographies and all markets (GSM, CDMA, 3G), while also offering vendors strong opportunities to differentiate.

However, the market is still barely in its infancy. The currently competing platforms – most visibly Symbian OS, Windows Mobile and Linux – have until recently been restricted to the high-end, less than 10% segment of the overall market. Symbian's strategy – of 'leaner, faster,

[5] To the surprise of some. But since homogenization in the phone context equals commoditization, it is perhaps not so surprising.

fitter' – is based on a drive towards the mid-range, boosted by natural momentum from increasing market acceptance plus a little help from Moore's law (which continues to drive hardware specifications up and BOM costs down, so narrowing the gap between the middle of the market and the top). Meanwhile, the incumbent operating systems in the mid-range are proprietary, for example Nokia's NOS/Series 40 combination of operating-system and UI or the operating system which drives Sony Ericsson phones. These can only be considered 'platform' operating systems in the sense that they support third-party programming in Java. It is in the mid-range that the next rounds of the battle for common operating-system platforms will probably be fought.

Looking further ahead, there are also possible competitive threats in the user interface area from alternative technologies which potentially offer UI alternatives either to the existing incumbents or other embedded operating system alternatives. Examples include building embedded device user interfaces directly in Macromedia Flash, more commonly known as a web-content display technology but proving itself as a UI-simulation tool and jumping the gap to become a potential user interface technology. Similar disruptive approaches include possible 'declarative' user interfaces based on XML-defined interface-description languages. It is not yet clear whether, or how, these technologies will eventually play in the market.

16.2 Background to the Eikon GUI

Symbian OS originates from the desire to create the perfect usable operating system for small handheld devices.

The original GUI – on the Psion Series 5 – was known as Eikon. Eikon itself, at least in the design sense, was an evolution of a previous generation of GUIs written for Psion's earlier 8-bit and 16-bit machines, although in concrete terms it was all new work, a second – or third or fourth, depending on where you start counting – attempt at mastering the GUI.

David Wood:

There were several UIs for the 16-bit software. In a way, it prefigures what's happened to the 32-bit software. For the MC400 which was a full laptop-sized device with a large screen, we designed what we called WIMP, standing for Windows, Icons, Menus, Pointer. And then for the handheld version we created something called HWIM: 'H' for handheld; we took the 'P' out because there was no pointing device. And then we did a new version called XWIM when we increased the screen resolution and it was actually binary compatible with HWIM, which again prefigures something that's happening nowadays, because the applications that ran on the original Series 3 also had to run on

later versions of the Series 3. So they ran in a compatibility mode with each pixel being doubled up.

The basic ideas about how a UI should work on a pocket-sized machine had been well worked through before any code was written for the Series 5. The iterative approach, however, was quite deeply a part of the Psion culture and scaled well to the size of the company at the time.

David Wood:

HCIL was the first version of the UI for the Series 5 and that was also quite WIMP-ish. For example, you used to tab your way around dialogs. However, after a while we decided that was too complex for most handheld users, so we abandoned having two-dimensional dialogs, we had one-dimensional dialogs which you navigated using the up and down keys. And that change led to the creation of Eikon, actually through two phases. After a while we decided we needed to refactor. After the first implementation we split it into two. We separated out the CONE control hierarchy as something that would be common for all user interfaces on the handheld devices because we viewed Eikon as just the particular interface for the Series 5. And CONE persists to this day. No doubt, it has evolved quite a lot but essential features of CONE are the same as in the 1996 version when CONE was first created.

From the beginning, even though the project was very specifically targeted at the Series 5, design decisions were taken with the later platformization of the operating system in mind, assuming a family of possibly different devices.

David Wood:

Splitting out CONE is a good example of the need to refactor. The need to refactor is an important principle that comes through architecting any large scale system.

A so-called 'Nokie' variant was created and the CONE classes were separated out. Eikon was re-engineered around this separation and the changes were then integrated back into a new version of Eikon.

David Wood:

'Nokie' is Eikon backwards. It's just the kind of black humor of development teams.

Eikon, in its time, was a complete concrete implementation of the Psion Series 5 GUI. Architecturally, it fitted into a complete UI framework comprising:

- Eikon: widget classes with look-and-feel policy and custom behavior
- CONE: generic control classes with no look-and-feel policy and generic behavior
- Application Architecture: the application model, broadly MVC-based,[6] although the detail of the implementation depends on the licensee GUI (UIQ3, for example, delegates most command handling to views).

This is a classically good object-oriented design, with two independent class hierarchies (Eikon and CONE) and an underlying framework exposing some generic behavior (common to all GUIs) directly to applications, with GUI-specific behavior brokered through Eikon.

16.3 Eikon Design Point

Windows, then as now, was the dominant consumer-oriented GUI when Eikon was first being designed, although both Macintosh and even AmigaOS (the Commodore Amiga operating system which included the Workbench GUI) had strong followings. (The line from Microsoft was then and still is 'Windows Everywhere!', from handhelds to data center servers.[7]) In the small device space, although Windows CE ran on some HP handhelds, there were other potential UI models too: Newton, for example, which was a strong influence later on the Palm UIs and PenPoint from GO! Corporation was an innovative and interesting, but commercially unsuccessful, UI. But as the adoption of MVC suggests, there were also other explicitly object-oriented influences.

David Wood:

Windows was one of our reference points, but we were aware of other user interfaces too. We saw what people were trying to do with Taligent for example, which was a combined effort between IBM and Apple that failed in the end but, like many failures, there were lessons to be learned from it. So we read avidly the books produced by people working on Taligent and we looked at that as a reference. Charles Davies was familiar with X-Windows, the Unix windowing system and that influenced us a bit also, though to be fair it more influenced the design of the window server than the UI.

[6] The model–view–controller pattern and framework is discussed in Chapter 14.
[7] See the comments in [Petzold 1992, p. 4], for example.

One critical design point for Eikon was usability – simplicity, naturalness and fitness for form factor of the GUI and the applications which shipped with it. Robustness and an intuitive user experience were the key principles.

Geert Bollen:

There was a logic which dictated that ultimately it is end-user benefit that is important, and therefore any abstraction at all can be broken for the sake of delivering something which delights the end user. And there was another important value, almost a law, 'Thou shalt not lose the user's data'.

Although Psion always practiced a strongly decentralized design regime with almost complete autonomy for the separate teams (sometimes to a fault), a lot of care went into ensuring that the shipped applications were consistent and well-designed from a user perspective. There was a UI Board, for example, which vetted the designs for all the application UIs.

Howard Price:

Bill Batchelor and Nick Healey would run the UI board and you'd go in and show them your design. There was a lot of attention to detail and pretty quickly they would be counting key presses, going down into the details of the look and feel. There was a lot of counting of key presses!

Another critical design driver was ease of application development.

David Wood:

One common principle behind the creation of Eikon was to make the job of writing applications easier and require less code, so that the applications would get more things for free provided they conformed to the framework. So that was a common theme: put the complexity in the system code and allow the applications to get the rich UI without having to do lots of detailed programming of that themselves.

The point perhaps needs some elaboration, because Symbian OS is sometimes perceived as being hard to write applications for. As discussed in Chapter 14, frameworks are a powerful object-oriented concept and the Eikon framework approach did indeed deliver a lot of power to applications. However, the Eikon programming model is considerably more sophisticated than a simple procedural one. Application developers have also recoiled from the complexities of the embedded systems-like

toolchain (develop under emulation, then cross-compile for what is, in effect, an embedded hardware target), as well as from the discipline required to develop robust software for devices on which a software error in an application freezing or crashing the system is simply unthinkable. For some, the development experience is a little too far from the desktop experience to be comfortable at first.

At the time, though, the big issue seemed somewhat different. Eikon was evolved together with the system software beneath it and the application suite above it. This approach continued the Psion tradition.

David Wood:

Eikon was created in an incremental process. As more applications were written I looked to see what the problems were that the applications had to solve so when people created a toolbar, for example, then we thought, 'Well, actually toolbars should be in the Eikon framework', and then people did more complex things with toolbars and we thought, 'Well, actually other applications would like to take advantage of this as well'. So systematically things that started their lives in applications moved into Eikon. And that made things a bit tough for the application developers because they had to rewrite things as we went along, but it did mean that with only a small amount of code in the applications, very rich user interfaces could be achieved.

The goal was to maximize the power of the framework to enable graphical applications to be developed with minimal new code and therefore to get the most from the small hardware footprint.

Of course, this didn't make things easy for the application teams working on the built-in applications. Peter Jackson recalls the frustration from the other side, compounded by the time-to-market pressures from the project.

Peter Jackson:

It was changing under your feet the whole time, and you knew that if you put effort into working on something that was based on that framework you were going to have to change it again.

Iteration was the natural model in the company at the time, and there was a long history of evolving from one product to the next. Evolving the Eikon GUI for the phone projects which were starting up in the wake of the Series 5 launch was in some ways just more of the same. Adapting the Eikon GUI to the very different form factor of each different phone was clearly a critical task and Eikon had been designed with adaptation in mind.

The reality, however, was that there was no clear understanding of how that should be done. Martin Budden moved from the application team to become technical lead on one of the very first phone projects, the Philips phone 'companion' (see Chapter 2).

Martin Budden:

We were still a way from forming a portability strategy for how we would develop and deliver the operating system for all these different manufacturers, so we were learning as we went along.

The Philips phone was the first adaptation of the Eikon GUI. The approach was straightforward and pragmatic, a straightforward branching of the components that needed to change. The Philips project developed a bespoke UI and a dedicated application suite with a small team, in not much time.

Martin Budden:

We did the UI but we also did the applications in there: we did the messaging, the contacts, and all those kind of things. To customize Eikon, we essentially rewrote the drawing code and the code for things that we needed to draw differently and we wrote new UIs for the new applications as well, using the underlying engines without change.

Murray Read joined the project in its early days, initially working for Origin, a software consultancy part-owned by Philips and later acquired by Symbian, but at that time supplying specialist software engineers to Psion.

Murray Read:

The UI design Philips wanted was quite different from the Series 5 design. It shared some similarities. I think we had a task list and the major parts of the UI were still there; there was a menu application; there was Window Server and CONE; and they were all there doing the same basic things. But when it came to the UI library itself, we had a much simpler set of controls to work with; one type of button, no keyboard, so we had to make the system work with the on-screen keyboard only, although it had handwriting recognition as well.

When the Philips project ended (disappointingly, without a product coming to market) most of the team moved on to start up the project for the Ericsson R380 and applied the same approach. Meanwhile the first

project with Nokia, for the new Nokia Communicator was in full swing (see Chapter 2).

Martin Budden:

The model of doing a bespoke UI was there, and then we did another bespoke UI for the Ericsson R380. The other big project at that time, going on in parallel, was the Nokia Communicator, and again, that involved doing a new UI to Nokia's specification. We swiftly recognized that there was a fundamental conflict between these UIs and it became clear that if we did a UI for every single phone that wasn't going to be sustainable.

The basic problem was clear enough.

Martin Budden:

We weren't thinking about generic problems. We were dealing with specific problems for specific projects that came up.

Each project was in effect a customization project, which created a complete, custom variant of the operating system as required for each device, from the base port to the operating-system services to the bespoke application suite to the UI, including modifications (at whatever level of the system) needed to support bespoke hardware such as dedicated phone keys or the R380 phone flip.

For all these challenges, the first phone projects were genuinely transformational for the company.

Ian Hutton:

The early projects were in fact extremely visionary, not just the Nokia Communicator, but also the Ericsson R380. The Ericsson R380 was a really advanced phone. The fact that it didn't sell in huge numbers, nor the Communicators for that matter, is largely irrelevant. There were really huge advances in design, both in what a phone should do, and how you could do those things on a phone.

Key features of the Ericsson R380 – the flip, for example, and its interaction with the screen modes, flip-closed and flip-open landscape – were good enough to be picked up by the next generations of phones and are still central to the design of phones such as the Sony Ericsson P900 series.

Ian Hutton:

The Ericsson R380 was very much an innovative design. It may have ultimately disappointed in terms of sales, but it did come up with a lot of good ideas and it solved quite a lot of problems in terms of turning what was then EPOC Release 5 into fully-fledged phone platform.

The project also contributed key technology back into the operating system. The View Server for example was originally developed by the Ericsson R380 team but has evolved into a key feature of the UI framework architecture.

Ian Hutton:

Initially it was used to solve a slightly different problem in the Ericsson R380, which was the flip mode. So it was initially written for flip open and, in fact, not just for the flip, it was for landscape mode too. It was written by Symbian's Licensee Technical Consulting (LTC) team for the customer project, and then the larger software engineering group was presented with it by the UI architects. Consequently, it was adopted back into the core architecture.

Accepting licensee (or LTC) changes back into the operating system baseline and evolving them forwards thereafter as part of the platform has been a central principle of the licensing model, right from the beginning. Making the mechanism work has not always been so easy, however.

Ian Hutton:

The issue is how to manage migration of changes initiated by projects to support project-specific and even device-specific requirements which then may have more general applicability and value, and migrate them back into the software base to avoid the base branching. So there's really a continuous process of splitting and some fragmentation, and then reunification – bringing it back together.

In part, the problem came from organizational tensions, the support for licensee projects being provided by a different engineering organization than the main operating-system engineering teams. Over time, the company has become adept at managing the process but there are still key hotspots around defect triage, change requests, engineering changes, and product requirements and the inevitable balancing of priorities between them.

16.4 The Device Family Strategy

Platformization of the operating system had always been a goal, whether or not it was a licensed platform or simply a Psion internal one. The licensing strategy was a logical next step. However, it is reasonable to argue that Symbian was not well prepared for the reality of running multiple licensee projects either in the practical sense of the straightforward logistics involved – it was highly difficult to resource and put immense strain on the company – or in the more narrow technical sense of how to manage the codebase-development practicalities.

Ian Hutton thinks that the Ericsson R380 project was significant in at least one other respect.

Ian Hutton:

Pretty much through the whole of that project it was difficult for LTC to work with the software-engineering organization. It was at the end of that project that the DFRD strategy was written.

For all the problems of branching and the resulting wrangles to migrate changes back into the main codeline, there was an underlying strategy emerging. It was eventually announced at CeBIT in Hanover in February 2000 as the 'reference design' strategy, based on so-called Device Family Reference Designs (DFRDs). As well as announcing a joint Motorola–Psion device, Symbian was showing off Ericsson 'mediaphone' prototypes based on what it called the Quartz DFRD – quarter-VGA, pen-based, PDA-style tablet devices with built-in phones and Bluetooth (enabling the use of remote headsets).

DFRDs emerged out of the need to resolve the problem of multiple, incompatible UIs. And it provided direction for Symbian and its engineering practice.

Martin Budden:

The problem was that it was not possible to come to agreement for a Symbian-based UI that was suitable for all parties. So this was when the idea of resolving all these conflicts with the DFRD approach emerged.

The engineering teams were at least in a position to abstract from the day-to-day problems of competing and conflicting projects and agree some higher design principles.

Martin Budden:

The idea was to have families of UIs. There would be one family which was based on the Nokia Communicator and the Series 5; there would be a family that was based on the Ericsson R380; and at about this time Quartz was started up for Ericsson, which was for the quarter-VGA tablet form factor.

The actual reference design specifications were based around a combination of screen size and orientation and input method. To an extent, the design points were really just rationalizations of the known preferences of the different licensees and were closely modeled on actual products that were already in development in licensee collaboration projects.

The DFRD strategy helped Symbian recognize that the phone world was complex and took a step forward from simply solving problems to systematizing the solutions.

The DFRD model set out to provide enough flexibility to support licensees across a wide spectrum from those, such as Nokia or DoCoMo, who had both the resources and the desire to create their own bespoke UIs, through a middle band of licensees who while not looking for a complete off-the-shelf solution preferred to work closely with a UI supplier like UIQ Technology than make the considerable investment to create their own UIs from scratch, to the smallest licensees who were looking for a near-complete solution and a fast product-development cycle. In effect, it reflected the standard tiering of phone vendors used by industry analysts:

- Tier One licensees create complete devices end-to-end: typically they expect to buy in Symbian OS and create or license bespoke UIs; they create lead products for new Symbian OS releases.
- Tier Two licensees create hardware platforms from standard parts: typically they expect to buy in Symbian OS with a pre-integrated UI; they create follow-on products from proven Symbian OS releases.
- Tier Three licensees focus on custom packaging of external phone design: typically they expect to buy in complete ('80%') hardware reference designs (from silicon vendors such as Intel and TI), with Symbian OS and their chosen UI pre-integrated onto hardware.

Bob Dewolf joined Symbian in early 1999 when Symbian acquired the Origin software consultancy. He had been working on embedded software for pagers and fixed-line phones.

Bob Dewolf:

I had just done a Farsi pager for Philips, and a smaller pager written in 16 KB of assembler, so that was quite the opposite of smartphones. But it wasn't a bad match because you had the embedded aspect from both pagers and telephone work and I had UI experience.

Symbian's DFRD strategy was newly in place. As a veteran of the early evolution from Eikon to what became Series 60, he knows as much as anyone, and probably an awful lot more, about the problems of crafting a phone GUI.

Bob Dewolf:

In that period between April and July 1999, we were in meetings with licensees, primarily in London, and they would get up and talk about their various plans for screen sizes and other key GUI design features. Those meetings were very, very important for Symbian to define a DRFD which fitted those users.

The first two DFRDs, known as Crystal and Quartz, were relatively straightforward to define since each in effect abstracted the actual properties of the particular target devices of projects which were well advanced. Each, in other words, had a single, clear customer. Crystal defined a keyboard-based device in the style of the Nokia Communicator, applicable to the Nokia 9210 and the Psion Series 5. Quartz defined a pen-based, tablet-style device applicable to prototypes that Ericsson (as well as other licensees) publicly demonstrated not long after. Between Crystal and Quartz, however, there was a missing form factor – that of a more or less conventional phone or, at any rate, a high-end device that was recognizably a phone in the sense in which the Ericsson R380 clearly was. The Sapphire DFRD was defined to meet the need of the third category.

Bob Dewolf:

The categories were QWERTY-keyboard-oriented, which was Crystal; pen-oriented, which was Quartz; and telephone keypad, which came to be called Sapphire.

Even then, the licensees who wanted to pursue designs in the Sapphire category had conflicting design philosophies.

Bob Dewolf:

One licensee would not be pen-oriented, while another didn't want to do anything without a pen. One licensee was very Java-oriented; another licensee wasn't. We talked about Blue Sapphire and Red Sapphire at that point, and I remember people trying to figure out how we'd have polymorphism. David Wood then produced some extremely interesting and abstract work about data and layout separation, and about abstract specification of components. I remember thinking, 'Yes, this is the way to do it', and saying, 'we shouldn't talk about those screen sizes, that's the last possible thing we want to talk about, we should be talking about the abstract definition of control sets!'

However, at the end of summer 1999, Sapphire was still blocked.

Bob Dewolf:

Then Nokia said they had a full design concept, and simply wanted to start developing. So one day about four people from Nokia arrived and we decided to see what happened. Originally, they were working from Sapphire documents, but not long after the name was changed to Pearl.

Out of the ashes of Sapphire, the DFRD that emerged was Pearl. By defining a new DFRD, the unresolved problems of Sapphire were circumvented. It seemed clear to those close to it that Pearl was strongly driven by Nokia.

Bob Dewolf:

Nokia had very strong time constraints. I remember sitting in meetings where we cut up responsibility for various types of things like soft keys and notes and queries and list boxes to various people and asked when they could get that done.

A lot of the main decisions for the architecture were made during that period. In fact it's surprising how many of the things we still have are based on decisions made in that period, things like doing all our multitap and key tries, like internationalization in the FEP (Front End Processor); using `CEikDialog` as the base class for notes and queries and forms, which you use in contacts; using the listbox-based classes, using the view architecture, and that was being pushed very strongly because of the Ericsson R380's success with the view architecture.

Some of the core architectural decisions were made at that point. It was a very interesting period. At the same time, the UI team itself was breaking Eikon down into Uikon, so they were refactoring and separating things out, moving things around, and we were trying to understand the customization

rules. Otherwise, at that point we were trying to stick within modifying C++ files in Eikstd and not touching Uikon at all. So we didn't do something new for notes, we didn't do something new for queries; they were still pop-ups. We were using the FEP architecture. We were very much trying to be a DFRD.

The DFRD approach required the separation of core GUI base classes that were free of a look-and-feel policy; the separation, in other words, of those classes that had no intrinsic look and feel and were therefore common to all DFRDs from the policy-laden, look-and-feel-bearing derived widget classes (which implemented the look and feel of a given GUI). The core classes would be part of the framework; each DFRD GUI implementation would create its own concrete derived classes.

The UI team in the software engineering organization therefore created a design proposal for a refactored Eikon to be called Uikon and the work was planned into Symbian OS v6.0, the first release which would support the DFRD model.

As David Wood sees it, this was very much a natural continuation of the evolution of Eikon, just as the earlier factoring out of the CONE control classes had been.

David Wood:

Just as we split out the control hierarchy and refactored Eikon, so later on people have sought to take other common elements into Uikon.

The original idea was really quite simple, based on a simple implementation strategy.

Murray Read:

The original plan was that everybody would just customize Uikon through what was called UikLaf – Uikon Look-and-feel. This was the DFRD strategy. Uikon provided the core UI, and Eikstd and any bespoke UI libraries were the customizable bits that you could modify. UikLaf provided an interface which you would use to modify the look and feel of Uikon itself, and it was a simple, flat, monolithic interface which would allow you to tweak various parameters inside Uikon.

But in practice, it just wasn't rich enough to give you the customization you needed to implement a UI like S60, for example.

Martin Budden is one of those who still feel that the Uikon architecture and the design choices that were made to support UI separation and customization were not always the best choices.

Martin Budden:

Uikon was an attempt to resolve the conflict between Eikon, which was the Psion Series 5 UI, and the UI developed for the Nokia Communicator. Essentially the idea was to add a further layer of abstraction that would allow both of those to sit on top of the same underlying UI. In effect, we ended up dropping Eikon, so that was how the conflict was resolved. Then Quartz used Uikon, but the so-called look-and-feel layers were just reimplementations; so the fact that they were in a separate layer didn't make any savings and you might as well have just reimplemented elsewhere.

What we could have done was standardize the APIs, rather than trying to make a split; standardize the topside and bottomside APIs of the UI framework so that you could replace it and still be compatible. That way you can run any application on any UI. You could slot in a different UI framework, instead of having a customizable UI framework and trying to separate out the customizable bits, which is problematic because there are no clear separations as to what is customizable and what is not. If you instead standardize the APIs, then you could just write a new framework to those APIs and things would just run.

Budden's views are based on his experiences during the Quartz project, for which he became the first technical lead, commuting between London and Ronneby.

Martin Budden:

The experience of doing Quartz highlighted that there was a lot of code that was not easily separable. For example, the messaging code had UIs in the engine layer, which meant that to change the UI we had to redo a lot of the engine code as well. There were lots of considerations that made it difficult to separate out the different bits. We rewrote a lot of the code, but without necessarily preserving the APIs.

The root of the problem was the UI fragmentation which dated back to the early days of the Nokia Communicator and Ericsson R380 projects. The lesson, Budden thinks, is a simple one.

Martin Budden:

In retrospect, we didn't do enough to support all those various UIs to ensure that they were following the same rules and were compatible.

Some degree of fragmentation was probably inevitable. Certainly, the risks of fragmentation were well understood on all sides and attempts

to avoid the worst consequences have been significant drivers of the UI evolution strategy. For example, the history of the Nokia–Symbian UI split shows a repeated pattern of branching and reconciliation of Nokia and Symbian UI APIs; S60 1.0 branched Symbian OS v6.0 significantly, leading eventually to the 'unbranching' Symbian OS v7.0s release, which restored API compatibility across the UI variants. Similarly, Symbian OS v9 has involved the unbranching of some UIQ APIs and will probably continue to include unbranching for both UIQ and S60.

Arguably, however, success is justification enough for the strategy that eventually evolved. With hindsight, different design choices could have been made; but getting devices to market is ultimately the test of success.

Murray Read:

My strategy for UI development has always been to make it work. I know people have this romantic idea that you can take one UI and turn it into another with a few tweaks here and there, but in practice to do that you've got to design that into the core UI in the first place. And that's not where Symbian started from. So my strategy has always been to just make it work anyway, write the new controls that you've got to write, adapt the existing code, branch things if you have to. And it's worked, I think S60 is the proof of that; we have a working UI in S60. And if we'd stuck to the strategy and followed the rules, I don't think S60 would ever have happened.

16.5 Quartz

The Quartz UI was developed at Symbian's Ronneby site in Sweden, which originally had been Ericsson's Mobile Applications Lab, a development lab working with Microsoft's Windows CE operating system. Ian Hutton moved to Ronneby to replace Martin Budden as technical lead on Quartz.

Ian Hutton:

Certainly with Quartz, I think the biggest challenge wasn't so much making that UI from the basis of EPOC Release 5, it was the question of working out what you want to do and then going out and finding something in Eikon to do it. The bigger challenge was really looking at the wider possibilities of the design, which was very challenging. A lot of that was focused around usability, what are people going to do with this UI. A lot of work went into that and working out what our underlying design needed to be to inform the UI design.

The view architecture inherited from the Ericsson R380 project, which had been migrated back into the Uikon framework, became a central feature of the Quartz application model, elaborated in the UI layer into a mechanism named direct navigational link (DNL).

Ian Hutton:

That whole view architecture, which is now in Uikon, was very different from what was being done by Nokia and it's very clever. The design of the views and using DNL was completely new – there's a whole new area of architecture there – and this behavior had to be defined and designed. That was one of the bigger challenges with Quartz.

Ian Hutton describes the DNL idea this way:

In the phone application, you've got your recent calls, you've got the callers' names, now you want to see the details of this person, so when you tap you switch to the contacts view, and now you can see that person's details, their phone numbers, email address, and so on. In the contacts view, when you tap on the phone number you switch to the phone application and it dials. So you've got some basic UI items and you want to make use of that in another application, and the underlying mechanism which switches you between those views is DNL which is quite a powerful part of the UIQ user experience.

In contrast, to do the same thing in the original Eikon GUI would require the sequence: go to Contacts; copy phone number; task away; find the Phone Application; paste phone number into the Phone Application; tap Dial.

16.6 Pearl

The Pearl team, meanwhile, was working flat out to meet Nokia's timescales.

Bob Dewolf:

The plan was to get the project up and running, and then, about six months later, not exactly freeze the APIs but exert more control over them. Then we would change the layout code so that it was resizable and more generic and customizable, putting in customization layers, and that would become the Nokia product. The DFRD would continue with that codebase and those APIs. But the focus at that point was definitely getting rapid development of their concept. However, there was increased pressure to change Uikon, and

I remember certain changes that were made at that point, we had meetings with the UI team and they said 'We can't do that. We've got no requirement to do that'. I guess there was skepticism about how we could customize this thing after it was created, and there was also I think a little pressure from the Interaction Design team, they weren't so keen on the Pearl UI.

The Symbian Interaction Design team was led by Scott Jenson who, before coming to Symbian, had been part of the original Newton team at Apple. Interaction Design had been closely involved in the design of Quartz. Pearl (or at any rate its concrete implementation) was a thoroughly Nokia-flavored UI, its design driven by Christian Lindholm and the team in Tampere (Lindholm recounts some of the history in [Lindholm *et al.* 2003]).

Bob Dewolf:

I remember Scott Jenson coming up with some criticisms of the UI. But you know, finding user problems with a UI is like shooting fish in a barrel, as they say, but Scott didn't like two soft keys, he thought it was too complicated. And then, out of the blue, there was a meeting and it was decided that there should be no UI in Pearl itself, Pearl would be UI-less. At that point everything changed. There was to be no UI deliverable from Pearl, and we became overnight the Nokia 7650 UI development team.

In a new shift of direction, Pearl was redefined to be a DFRD without a user interface, or what came to be called a 'headless' DFRD. In effect it was the first step towards the post-DFRD strategy of a headless operating system. The Nokia 7650 (arguably the first true phone based on Symbian OS) launched what was branded at the time as the Nokia Series 60 UI. Pearl thus became Series 60. Dewolf sums up the situation.

Bob Dewolf:

Crystal was a product which was elevated to a DFRD. We had a DFRD which was deflated to a product.

16.7 Nightingale

Series 60 (now rebranded as S60) was announced at Comdex, Las Vegas, in November 2001. The Nokia 7650 itself (rumors of its launch had been circulating for several months[8]) caused quite a ripple of interest when it

[8] For example see online articles in the Register around November 2001.

was shown a few days later at Nokia's Multimedia Developer Conference in Barcelona, before its launch in February at 3GSM 2002 in Cannes. It was the first product on Symbian OS v6.1, the first Symbian 2.5G (i.e. it supported GPRS) phone and, with its VGA camera, it was the first camera-phone anywhere outside Japan. At that time, the Nokia 9210 had just become the best-selling PDA, unseating Palm. Symbian OS rose for the first time to the top of the platform chart for PDAs and smartphones.

Bob Dewolf:

Series 60 made us think about all the things Symbian OS had to worry about, like distribution policies and whether components are only published internally or were third-party published. We put in a whole new layer of control to say not only that header files export but whether they get filtered out to the SDK process. Pearl became Series 60 at that point, version 0.9, and SDKs came out soon after at version 1.0.

But above all, the Nokia 7650 was the first S60 phone. The big announcement at 3GSM in Cannes was Symbian OS v7.0. At the same time, Symbian announced a new UI strategy. The DFRD idea was dropped. In fact, Symbian no longer proposed to ship a GUI implementation at all on top of its UI Framework implementation. The field was opened to UI vendors or licensees to do their own thing. Meanwhile the new UIQ UI was launched in place of Quartz, with UIQ spun out as a separate, independent company, though it remained Symbian owned. (In late 2006, UIQ was purchased by Sony Ericsson and became completely independent of Symbian.)

A few weeks later, the first Symbian OS v7.0/UIQ phone, the Sony Ericsson P800, was announced (it launched later that year). It was a pen-based phone featuring a removable flip keypad, somewhat in the style pioneered by the Ericsson R380, a jogdial thumbwheel for navigation and, of course, a camera.

The new strategy was possibly the only strategy that made sense not just of the unique nature of the phone market, but of Symbian's unique position in it. Perhaps, to some extent, Nokia identified this most quickly (and so platformized its own UI). The moral, perhaps, is that where it is impossible to have more than partial foresight of evolving market requirements and opportunities, multiple visions are better than one. Enabling a competitive ecosystem allows multiple different visions to emerge and have a chance to succeed.

The change in strategic direction predated Colly Myers's departure as CEO in February 2002, but its implementation spanned the interregnum and was picked up and seen to fruition by the incoming CEO, David Levin, from April 2002. Fittingly, Myers's – and Nokia's – Christmas gift to the company was a Nokia 7650 phone apiece.

16.8 How to Develop a World-class GUI

The most visible mobile phone trends are in conflict: devices are becoming increasingly smaller, packing in more and more hardware and demanding longer battery life, while moving increasingly into the consumer end of the market as features and functions expand. The phone operating system is the battleground on which these conflicts are played out. Shrinking size and increasing complexity are not simple trade-offs. The evidence is that the successful phone needs both.

Perhaps it is worth citing the definition from Pekka Ketola, a usability engineer on S60 [Lindholm *et al.* 2003, p. 172].

> Smartphones are an emerging product category where communication, namely voice calls and messaging, is still the main function, but where personal information management is fundamentally improved compared to conventional mobile phones. Smartphones have good calendars, versatile contact management properties, to-do lists, address books, and so on. They are solid platforms for imaging and gaming.

In one way, phones are become increasingly personal items, statement-making lifestyle accessories. In another direction, they have evolved from being 'expensive showoff tool[s] to an everyman's communication platform' (Nieminen-Sundell and Vaananen-Vainio-Mattila in [Lindholm *et al.* 2003]). In both directions, the trend is to a normalized market, that is, a consumer market in which sophisticated users give way to naïve ones and complexity is not tolerated. Phones lose novelty value and are expected to perform as easily and predictably as TV sets and stereos. In this battleground, the UI becomes the front line.

Big User Interfaces Don't Scale

Bob Dewolf:

There's an awful lot to a UI. It's not just controls and their customizations, there's a lot of interaction. That's a surprise we got in the process of development when we did Pearl, how much more integrated you had to be. Integration is how the applications fit together, what keys the phone is grabbing from the window server, and who's managing the power key and what happens if you press the Phone button. There were a lot of issues like that and it was important to get them right.

I remember very fundamental decisions about what happens when you press the menu key. For instance, does it just put the menu on top? In the we decided on something very simple, but there were lots of different proposals,

which led us to decide that simplest is the best and we just leave everything stacked where it is and go with the window-server policy. But it's amazing how long the debates took. In that particular case, it ended up very simple, but not in all cases.

An absolute rule is that small interfaces are different from big ones. Direct manipulation (the familiar Windows or Macintosh model), for example, does not scale down. You do not expect to drag a mouse cursor across your phone screen. The parallel model of the desktop, with multiple open windows between which you task, does not scale either. On a small phone screen parallel is out, sequential is in.

Just as even the best UIs do not scale, nor do they move easily from one device class to the next. In effect, this is exactly what Symbian's UI strategy reflects and what the architecture has been evolved to support. One size does not fit all, and there is no single right model. What works on a flip phone probably does not work on a keyboard-centric Communicator; what works on a pen-driven tablet probably does not work on a phone designed for the shirt-pocket.

The first generation Quartz devices (that is, before the evolution of Quartz into UIQ) were strongly tablet-based designs. There have been tablet phones (for example, the original XDA based on Microsoft Windows CE and, perhaps, the phone-enabled iPaq counts as tablet-like) but they have not been huge successes in the mobile phone market. The evidence suggests that consumers want phones with productivity functions, rather than PDAs (which are really productivity devices) with phone functions. There is little to suggest that the UI is a critical factor in the decline of the PDA market; neither Palm PDAs nor Microsoft PDAs have fared much differently. (Palm has now gone all the way and adopted Windows Mobile, so the point becomes moot going forward).

Given the differences between Windows on a desktop and Windows Mobile on a phone, there is not much case to be made that users are seeking the desktop-style behavior on their phones; they won't get it. What they will get is familiarity at some level, even if the behavior is different. Perhaps what they are really seeking is the unstated promise of compatibility; but in that case the value of having Microsoft running on their phones is only incidentally about the UI itself and has more to do with the promise of the brand. But as the Palm and Macintosh combination demonstrated quite ably years ago, smooth interoperation has not much to do with sharing a UI on different platforms, it's just a matter of plain old-fashioned good, careful, user-centered design.

The most interesting point about phones is that while the design point has in a sense been fixed and is the same across manufacturers, UI differentiation makes different phones quite radically different from each other (to the point that picking up someone else's phone and trying to make a call can be surprisingly difficult). Kiljander and Jarnstrom (in [Lindholm *et al.* 2003]) argue that it is because phones have no standard UI and, therefore, everyone is using their own solutions that the UI has become such a significant competitive asset. This is as true at the low end of proprietary operating systems and custom UIs as it is at the high end of Symbian OS and its competitors.

If this is the case, Nokia's commanding market share (at the time of writing anyway, but with little sign of it waning) is a testament to its UI designers and to its strategy of evolving a family of UIs, each tuned to the design point of one of the categories in the famous Nokia segmentation matrix.[9]

Usability Values

As well as becoming cameras, music players and diaries too, phones have become open platforms for third-party software of all kinds, including games. It is hard to devise a UI that is as fit for playing fast-paced shoot-'em-ups as it is for managing your daily, weekly and monthly meetings and appointments, while still allowing users to answer (and initiate) calls with a single button press. While phones have become the archetype of the omnifunctional, converged device, Symbian OS is still designed to be flexible enough to support a wide range of possible device types including more narrowly specialized categories – digital cameras, in-car navigation systems and set-top boxes have all been rumored at one time or another as possible target devices.

Arguably, however, the particular, and unique, strength of Symbian OS is its power (as well as its compactness) as an open, thoroughly GUI-centric, standards-driven application platform. The fact that phones are no longer merely phones, but complete open software platforms is a critical turn on the road. It is also a central part of Symbian's play for the market. Applications are critical.

The Application Model

A key usability issue is that of the application model exposed to users: how does the user work with files, documents and applications, and what

[9] Nokia segments its users according to customer categories (Balancers, Controllers, Experiencers and Impressers) and its phone models according to style categories (for example, Expression, Classic, Fashion and Premium). The fully populated matrix matches phone styles to customer segments. Typically, Nokia's different UIs (Navikey, Series 30, Series 40, Series 60 and Series 80) match their complexity to the different requirements of the segmented model. See the chapter by Kiljander and Jarnstrom in [Lindholm *et al.* 2003].

concepts (and how many and how intuitive are they) must the user grasp in order to use the device successfully and naturally.

Andrew Thoelke:

Windows has never really ever escaped being file-based. You think in terms of files. Everything is a file. Macintosh at least went one step further because you generally didn't find files: you had documents and you had applications. The Series 3 had a system screen that was application-centric and a file browser that gave you a folder view of everything. So the system screen showed you what we called the washing line of applications, and under each application it said, 'Here are all the documents this application knows about'.

With Series 5, we went to a bolder view of the world. It was document-centric, not application-centric. You could basically tap on a document and the system recognized the document. It knew exactly what application was associated with it, and it would then launch that application and give you your document. We didn't believe in file extensions as a way of identifying content, so we invented our own system of internal designation, UIDs (Unique Identifiers). Macintosh, by the way, has always been the same. But our application architecture in Symbian OS really came about from saying, 'Okay, so given a document I need to basically do something with this document', and that's what the user really cares about. The fact that there was an application doing it for you was almost like, 'Well, the user shouldn't have to worry about that. The user shouldn't have to care'. Of course you knew that if you hit the Word button it would take you to that application, but if you found a document that was just sitting there you could tap on it and ta-da! It would have the right icon and it would just all work. Well that is probably right for a PDA, because a PDA is all about creating documents. But actually that isn't right for a mobile phone.

The native, document-centric application model also runs into trouble as devices start to become increasingly connected. While documents created on the device, by the native applications, can be created with appropriate UIDs which identify their document types (stores and DBMS databases, for example), recognizing externally created documents so that the appropriate application can be launched is more problematic. The early use cases around which the Series 5 was designed were manageable, mostly requiring recognizers for email and web document types. But in a modern phone the sheer diversity of the document types which may be encountered, ranging from office-type documents to a wide variety of media formats, requires not just a comprehensive recognition system (which Symbian OS provides), it also forces at least a degree of file centricity on the UI. Thus UIQ, for example, which has a non-file-based application model, must adopt a file-based approach for dealing with media files. S60, which is file-based, runs into a slightly different problem when swapping files between devices because the built-in file manager

application chooses to hide significant parts of the file system for usability reasons with novice users. (A third-party file manager is required before the user can fully explore the file system.)

In Symbian OS, the application model is, in effect, created as a collaboration between the operating system and the UI implementation. In the original architecture and in the Uikon architecture which was derived from it, the application model (i.e. the document framework) is implemented by the application framework but brokered through the UI framework (Uikon) and the UI layer which is built on top of it.

Both UIQ and S60 have evolved away from the original application architecture, for example of the original Eikon GUI, towards more task-based approaches. Underlying that evolution is the difference in the way that PDAs and phones are used.

Andrew Thoelke:

If you're using a mobile phone, the way you think is very much doing one thing at a time. The fact that a Symbian phone can multitask is something that probably most people don't realize, except that as soon as you go back to an old phone you realize that when you send a message you've got to sit there waiting for it to be sent. On an S60 or UIQ phone, you send a message and you carry on doing everything else, and eventually the phone says, 'Oh, by the way, I've sent the message'. So the multitasking in a phone is a bit less obvious. Generally on a phone you are task-oriented, so you're interested in what you're doing right now, and then in the next thing you're doing, and so on. Some UIs are better than others at being able to backtrack to what you were doing just before you interrupted it, but in that respect you really want to have a system that is based around applications and tasks. So that's where UIQ has gone, in the direction of having the view server, where a view is essentially a task, but sometimes it's one way of looking at tasks. So each application has got different views and you can flip between those views, but also flipping between views will activate different underlying applications. Whereas S60 has a quite different model of either embedding views or just jumping between applications, and it produces a different system.

In its time, the Eikon approach was quite innovative, and it is part of what made the Series 5 so easy to use.

Andrew Thoelke:

I guess it wasn't innovative in the software space in '94, but certainly in the PDA space, to have something that was document-centric and so highly integrated with the actual application framework. It was certainly unusual. And it has not been a barrier for being in the mobile phone space, but it's not

necessarily been totally tuned to the way mobile phones work, and the UIs have had to address that.

Java proved to be another area in which the cleverness of the application architecture and its seamless integration into the UI caused some unexpected headaches. Java proved quite hard to integrate well with the subtly different application model of Symbian OS, although for the MIDlet writer, as for the user, the differences are transparent and Java applications 'just work'.

Andrew Thoelke:

Even today the system still thinks you're a native standard Symbian C++ application, so for Java we have to provide special 'stubs' and 'dummy' resource files to make the application architecture think its got a valid application to run. It would be better to be able to register with the application architecture and tell it there is an application and how to run it. That would be the ultimate, to have an application architecture and an application lifecycle model that is truly agnostic to the actual programming language.

16.9 Symbian OS User Interface Architecture

Symbian OS supplies the UI Framework and Application Services as its topmost layers, but does not supply the concrete user interface. Instead, phone vendor licensees create (or license) a UI which is based on the framework classes supplied by Symbian OS. Symbian OS has no notion of a console application. Native applications, by definition, present a UI which is implemented using the concrete classes of a particular GUI implementation, layered over the generic GUI framework support made available by the operating system. Applications implement the abstract classes of an object-based, MVC-style, event-based application model which is defined partly by the operating system frameworks and partly by the GUI implementation. In fact, the event model goes surprisingly deeply into the system, all the way to the Window Server.

Non-native applications, written in Java, Visual Basic, OPL or, more recently, Python and even Ruby (these are currently the most mainstream high-level languages available on Symbian OS), depend completely on the graphical support of the GUI implementation on top of which they are written. Except for low-level development work, for example system programming, writing Symbian applications is inherently GUI-dependent.

Symbian ships only non-production test application UIs for system components that require a UI for testing, configuration, and so on. In a

final licensee product, licensee applications manipulate the components directly or provide the application-level user interface.

16.10 Future Directions

From 2006, smartphones are shipping with the latest releases of both the UIQ and S60 user interfaces, both of which introduce new features to support UI differentiation and specialization, aimed particularly at enabling phone vendors and operators to customize, extend, and brand phone UIs. In both cases, the latest UIs depend on the Symbian OS v9 platform.

If these releases are indicative, increased flexibility and customizability certainly seem to be the dominating trends for the future evolution of phone UIs.

UIQ 3

UIQ 3 emphasizes personalization and configurability. It includes features that offer extended flexibility and new opportunities for differentiation and specialization by phone vendors and operators, as well as new opportunities for third parties to provide end-user-installable UI customizations. In particular, UIQ 'themes' bring the notion of the skinnable user interface to UIQ. Themes may be ready-made (in other words, supplied by UIQ or created or customized by the phone vendor or other UI integrator), user-customizable, operator-specific or supplied by a third party, encouraging the market for downloadable themes.

Operator customization based on Operator Configuration Package (OCP) themes and skins allows the provision of preloaded content including applications, sounds, settings, fonts, icons, animations and so on, by vendors supplying UIQ phones to operators. There is also support for over the air (OTA) features, allowing operators to configure (and customize) phones remotely.

Supporting these features is the notion of parameterizable UI properties, with support for multiple screen sizes, multiple portrait and landscape modes, touch-screen and non-touch-screen modes, and various hard and soft key combinations. The resulting UI configurations (for example, flip-open and flip-closed styles and pen and softkey styles for different product families) are intended to help developers maintain application compatibility across a range of different devices while enabling vendors to create a wider variety of phone styles.

S60 3rd Edition

Again, much of the focus of the S60 3rd Edition UI is on enabling customization, including vendor, operator, end-user and enterprise customization. Increased flexibility is emphasized too, with UI scalability

supporting a wide range of screen sizes, both portrait and landscape modes, and 'double-resolution', that is, high-resolution displays.

The most significant support for differentiation is the enabling of core application extensions, allowing operators to extend the core applications with new features while retaining basic application compatibility with other S60 phones. The emphasis is on supporting integration of operator-specific features and services, as well as service settings and operator-customized UI look and feel, to enable a high degree of sophisticated operator-specific branding to be provided by phone vendors using S60 on their products.

17

System Evolution and Renewal

17.1 Introduction

This case study looks at the forces and pressures that have driven the evolution and renewal of Symbian OS, from the earliest days of the system to the latest Symbian OS v9 releases.

A critical issue for any software company is keeping a complex software system fit for purpose in a context of rapidly changing technology. To do so requires continuous evolution and continuous renewal. Evolution and renewal are, therefore, among the strongest internal drivers for engineering and architectural change.

Customers, on the other hand, are typically more interested in features. Balancing the competing pressures, for feature growth on the one hand and internal renewal on the other, is an important part of the business. Preserving the freedom to invest in renewal against relentless feature pressure and deciding where to focus that investment are critical elements of a successful system strategy.

In reality, of course, the two things are not independent. Renewal is just another way to talk about the prerequisites for future features. But, simplistically, while the costs of new features might be charged to customers, the costs of renewal are an overhead. Left at that, features tend to win the competition for development resources.

Symbian OS faces the particular challenges of the mobile telephony context, which are acute. The pace of evolution since mobile telephony began to take-off in the mid-1990s, and particularly since the second boom around 2000, has been breathtaking. That period of a little under a decade is also the period during which Symbian OS has been licensed to phone vendors. The increase in capabilities of Symbian OS v9 compared with the original release of Symbian OS is quite dramatic.

Charles Davies is forthright about the need for continuous renewal.

Charles Davies:

All of Symbian OS needs to be renewed over time and it's a challenge to work out how to do that. In architectural terms, renewal is one of the biggest challenges.

In theory, five years or so from now, Symbian OS will be at the outer limit of its original intended design life of approximately 15 years (depending on when you start counting). Of course, it is a fact of life that software systems commonly outlive their design lives by several multiples. Bits do not wear out. Software does not fall apart over time. But still, the changing requirements and changing context mean that, in practice, system evolution is almost continuous. The likelihood is that, in five years time, most of the headline features of the operating system, and many of its significant architectural features, will be ones that were not present in the original releases; many of them may not even be present today.

This is the magic of renewal (or the problem, depending on how close you are to it). Unlike the products which are built on top of software systems, software design life is typically managed by forestalling obsolescence through continuous re-invention and evolution, rather than by withdrawal and replacement of products (although that difference may have much to do with the less tangible nature of software compared with hardware).

17.2 Design Lifetime

All designed systems have a design lifetime and, for any manufactured system, deciding what that lifetime should be is as much about economics as it is about technologies. For example, the design lifetime is one determinant of cost because it determines quality requirements for components. It is also a key variable in calculating return on investment and other accounting metrics and is, therefore, a key variable in deciding the business case for a system and it remains a factor in deciding how much to invest in maintenance.

Software systems are no different. However, unlike other manufactured systems, they do not wear out. Maintenance is not driven by use of the system causing parts to become defective because of 'wear and tear', but by change. Maintenance in a software context mostly means defect fixing. Typically, the defects to be fixed are either those introduced by earlier (internal) software changes, or those which are revealed by an (external) change in the way the system is deployed. Thus, software maintenance is driven by change, whether adding new components to the system,

removing components, or using the system in a new product, or because some other contextual change exposes previously ignored defects.

A simplistic way to distinguish maintenance from renewal is to count changes made to keep a system viable during its design lifetime as maintenance and changes carried out to extend a system's design lifetime as renewal. It is because software systems do not wear out that renewal is possible and attractive. But increasingly, what makes renewal necessary is the rapid pace of technological change. Maintenance, therefore, shades into renewal and is necessary not to extend system design life but to ensure that a system actually survives for its design life and is not made obsolete by technology changing more rapidly than predicted around it or by new, disruptive technologies appearing and changing the external context to one in which the system cannot compete. Renewal becomes as much a matter of survival as of extending the design life.

Deciding between maintenance and renewal is by no means always easy. There is a critical balance between maintenance and renewal, and getting the balance right can be a tough challenge for any organization; deciding when to maintain and when to renew requires foresight, a clear and well understood strategy, and a good understanding of the technology context. In the short term, maintenance always looks attractive; renewal frequently looks expensive.

Determining the Design Lifetime of Symbian OS

There was never any doubt about the goals of the software project for the Series 5 operating system. It remained as it began, focused very clearly on the requirements of that very particular device. However, there was also a broader context.

Geert Bollen:

There was already explicit intent to create an operating system suitable for multiple mobile devices, with a design life of 10 to 15 years. There was long-term thinking and there was also a vision of having software licensees that included mobile-phone manufacturers.

David Wood is equally clear about the long-term nature of the original vision.

David Wood:

I think we said something akin to 'The 16-bit system was designed for five years and would last ten; EPOC32 was designed for ten years and would last for 20'. It is interesting that we have already been through ten years of history, and I think it's going to be around a lot longer than even we foresaw.

Even perfect foresight would not be enough to secure a 15-year system lifetime. There is no option about accepting the challenge of renewal. The need for renewal is a fact of life. David Wood argues that it has been fundamental to the way Symbian OS has evolved right from the beginning.

David Wood:

The fact is that we did redesign many things in the initial evolution of Symbian OS. The UI is an important example of that, the fact that we went through three separate UI designs to get to the Eikon UI. And there were similar changes, not quite so large-scale, but other changes in the implementation of other aspects of Symbian OS, even before it got launched.

While you cannot guarantee longevity for a system, you can certainly design for it. Designing for longevity means designing for the unknowns to which your system will be subjected in the future. What that really means is designing extensibly.

The two prerequisites for extensibility are a good conceptual design and natural extension methods. The Symbian OS architecture is built on multiple levels of extensibility. System services are provided by servers that serialize access to system resources. To provide a new service, you just create a new server. Most servers are themselves extensible, because they are implemented as frameworks. To extend a server, you just create a new plug-in. Within the system there are multiple patterns of abstraction. Active objects, for example, abstract all events, so that applications only need to know how to handle the events they care about, without worrying what their source was. Sockets abstract all communications bearers, so that applications only need to know about the message types they want to send, without worrying about message transport.

These are all examples of extensible design and there are many others. David Wood tells a story to illustrate his point about the way extensibility is designed into Symbian OS at a deep level.

David Wood:

Bluetooth only emerged as a technology in 1998, about the same time Symbian was formed. So when we designed the system, Bluetooth didn't exist. Now I'm not trying to diminish the efforts of Symbian engineers, but when we did want to implement Bluetooth it wasn't that hard. Of course there are many devils in the details but architecturally it slotted in quite easily as just another active object, just another event source. In broad terms, it fitted easily into our architecture; we had foreseen that need by designing to be future-proof.

At one of the early developer conferences there were questions from the floor during a talk on implementing Bluetooth on Symbian OS, asking, 'How

did you manage to do this?' When the answer was given, the question came back again, 'But surely it couldn't have been as easy as that!' And the answer was, 'Well, for us it's just another active object.' That was the phrase used: just another active object. Later on it turned out those questions were from Palm engineers, from the operating system part of Palm, and clearly they had been struggling to squeeze Bluetooth into their system which hadn't been sufficiently future-proof. In the end they managed, of course, and there certainly is Bluetooth on Palm OS today, but it required much more effort I think than with Symbian OS. So I do think Symbian OS is future-proof, because we design our frameworks with extensibility in mind.

'Future-proof' is a bold claim but it was certainly the goal of the early design of Symbian OS and, in key respects, has demonstrably been achieved.

There is, perhaps inevitably, danger too in abstraction. Too many levels of abstraction can lead to code bloat and poor performance and also to over-complex and unmaintainable code. The job of the component designers and architects is to understand where extensibility is required, and to enable it appropriately, and where it is superfluous and constitutes over-design and over-engineering.

High Impedance of Change

Technology moves fast and each technology generation seems to move faster. The founder of Netscape, Jim Clark formulated his 'Internet-time' law to express that fact, the so-called law of constantly increasing acceleration which compels 'technological products to be released faster and faster' [Lewis 1999].[1]

However, systems, especially large and complex systems, evolve slowly. This is not just true of software. It is as true of social organizations (from large bureaucracies to large crowds) as of whole species (evolution) as of particular technologies (fossil-fuel-powered systems). Thus the corollary of the speed of change of technology is the high impedance of systems to change.

Entrenched systems change most slowly, as if large-critical-mass systems are slowed by systemic inertia. Operating systems become entrenched when they achieve a certain level of market saturation. This kind of entrenchment and inertia may be the biggest threat to renewal.

Two ideas which have become central to what Symbian does, and how it does it, reflect the interesting dynamic between the drive for change and the drive for stability:

[1] The limit is based, as he says, on the speed of light: 'How fast light moves down a fiber-optic cable.'

- Platformization is what drives Symbian's business and technology goals, the goal of creating a software platform suitable for hosting end-user-centered services across a diverse range of phone hardware
- The lead product concept is a key part of the way Symbian drives projects, so that each software iteration is closely coupled to and driven by the requirements of a particular licensee product.

Platformization requires stability. Lead products, on the other hand, drive innovation and change. In part, this approach has crystallized naturally, out of practice.

Providing an open platform for third-party development and bringing it to the emerging world of mobile phones (and other pocketable devices highly centered on communications) was one of the early rationales for Symbian OS, and remains as much so now as then. To make that strategy support multiple licensee UIs (to enable licensee differentiation), an equally basic rationale, requires a high degree of commonality in the system below the licensee UI variations (80:20 was the early rule of thumb guideline: 80% of common APIs underlying 20% of UI-specific APIs). The challenge of the strategy, therefore, is to avoid the platform fragmenting beneath the centrifugal effect of UI variation, to enable as far as possible a write-once–run-on-all-variants development model. Some customization for UI variants will always be required, but the minimal goal of common data engines beneath custom UIs is achievable so long as unnecessary fragmentation is avoided.

The lead product approach has proven a successful way to capture new requirements and drive the evolution of Symbian OS forward while supporting licensees in bringing a more or less continuous stream of leading edge products successfully to market and, in particular, while maintaining the hard product deadlines required for manufacturing.

All Symbian OS projects are tied to one or more particular lead products that define the basic critical feature set for a release, drive the release schedule, and provide early validation and real products for final testing. Once the lead product has shipped, licensees can consider that OS release a proven platform for subsequent product variants. In effect, this simply rationalizes the early approach of working closely with licensees on real products, but provides a framework for supporting multiple simultaneous licensees taking a given new release.

17.3 Renewal in Symbian OS

Typically, a renewal strategy aims to anticipate future feature requirements and evolve the system to be a better platform for those features than the

existing platform. To a lesser extent it also aims to improve any areas where the platform is deficient for current features.

Renewal permeates Symbian OS. Examples range from major system-wide changes, which include UI evolution, the introduction of the real-time kernel, and platform security, to numerous small upgrades and updates, for example to track standards evolution in supported technologies (including Bluetooth, SyncML, vCard and vCalendar). Between the two extremes are many examples of significant architectural enhancements, whether to evolve services such as telephony to support new network technologies, to introduce new services or to replumb essential system services.

- The UI architecture evolved through multiple iterations to the early Eikon UI and, thereafter, to Uikon plus licensee UI variants (for example, S60, MOAP and UIQ), which are themselves continuously evolving.
- The new kernel provided a complete generational change at the heart of the operating system.
- Platform Security was a major architectural evolution with system-wide impacts and high external visibility.
- Large-scale tools and system infrastructure changes include toolchain upgrades from Visual C++, to Codewarrior, to Carbide and ARM, tools updates driven by the move towards standard C++.
- New services include the power-management and system-startup frameworks.

There are numerous examples of small architectural changes related to the above major changes, including evolution of the IPC mechanism, a new client–server architecture, and the ECom plug-in framework architecture. Other examples of significant architectural evolution include two generations of multimedia re-architecture, significant telephony evolution to support new network technologies (CDMA and 3G) and a forthcoming new communications architecture.

There are also numerous examples of continuous architectural evolution driven by performance requirements and of extending existing services as new technologies emerge, for example, flash filing systems and USB support.

The largest changes have been phased in over multiple releases, partly to manage the risks to licensees of large-scale change and partly to try to maintain an incremental approach to development and to maintain the model of frequent releases. (Symbian's earliest releases were monolithic and the project management lessons were learned the hard way.)

For example, the real-time kernel was introduced as an option for Symbian OS v8 and then became the standard kernel for Symbian OS

v9, enabling licensees to evaluate the new architecture and giving them flexibility in when to migrate. The introduction of system-wide platform security was similarly phased in over two v8 releases: v8.0 and v8.1. (It overlapped with the introduction of the new kernel because it depended on the new kernel to police the security policy implementation.) Both are examples of a major re-architecture of the system and both cases reflect Symbian's perception of the needs of the market. Interestingly, in both cases there was some short-term customer resistance, even if customers have later embraced the changes.

David Wood:

It has taken a long time for EKA2 to come to fruition, but that has been an internal renewal of Symbian OS and that is an extremely important initiative that's happened, because it has addressed some of the things that, over the course of time, it became clear to us were shortcomings in the original design. So it's very good that we've been able to do that. I don't think the full significance of the new kernel will be appreciated for some time, but we will see it employed not just in smartphones, but in a large number of other mobile devices too in my view, because of what we were able to do to refresh the design.

Platform Security is a highly significant change, which introduces a security capability model and data caging supported by a certification scheme, and requires some significant kernel changes including changes to IPC. The overall impact has been high and, in platform terms, is highly visible, because it has immediate (external) developer impact. Again, the project had to weather some significant early resistance. To some extent, the external impact may not yet have been fully felt. Charles Davies is sanguine.

Charles Davies:

It's a big thing, you know. There is no industry-wide accepted best practice for how to secure a system, I think nobody knows exactly what the best thing is to do. There is no 'right' answer. But PlatSec is our answer and time will tell.

17.4 Evolution in the Kernel

The move to the new EKA2 kernel is perhaps the most significant evolutionary step Symbian OS has taken. However, as Martin Tasker points out, architecturally it remains in some ways quite a local change.

Martin Tasker:

At the system-design level, it hasn't actually radically changed the system design. It's still either application processes or server processes and that design was pioneered way back in SIBO and hasn't changed much since the earliest releases of EPOC. One reason it hasn't changed much is that it's a proven design.

In fact, the beginnings of the kernel re-architecture go all the way back to the first collaboration with Nokia on the project to bring the 9210 Communicator to market. This was not Symbian's first licensee project and it was not the first Symbian OS phone project.[2] However, at the time it was certainly the largest and most complex Symbian project to date.

The design approach was based on a two-operating-system, two-processor solution, the so-called 'partner operating system' approach. Symbian OS, since it was not then real-time-capable, was not capable of hosting the GSM telephony stack. The phone side and the application side were therefore separated, with a dedicated RTOS running on a dedicated phone-side processor and hosting the baseband software, while Symbian OS hosted the application side, running separately on an application-side processor (see Chapter 15). While Symbian OS has evolved hugely in the intervening years (probably almost as much as the phone hardware has evolved from the first, brick-like Communicator design to the latest sleek devices), the problems of that early solution are instructive. The most immediate problem was how to force the phone-side RTOS and Symbian OS to cooperate.

Morgan Henry:

To integrate the two OS schedulers and interrupt handing, there was only a small amount of management that EPOC did first and then it would call one of these hooks to hand-off to the RTOS. The RTOS would only give control back once it felt it had done its job. I believe there have been similar solutions for things like Linux running with a partner RTOS. But the challenges aren't over yet and you have to decide what owns which bit of hardware. Some of the hardware was owned by the RTOS and some of it was owned by EPOC. I think the MMC card, for example, was owned by EPOC and all of the peripheral drivers, but all of the baseband hardware was on the RTOS side, so it owned the power management, and it owned the real-time clock which caused some interesting problems.

Coming from the Series 5, where EPOC always had direct access to every bit of hardware, especially the real-time clock and the timers, and now finding

[2] The Philips Ilium/Accent and the Ericsson R380 projects both preceded it, though only the R380 came to market.

that it had to go over a communications protocol to get the time from the real-time clock was a bit of an architectural shock for it. So there were lots of work-arounds, but where we had the opportunity to do it we tried to redesign in a sane way. This is what started the move to try and push a lot of things outside the kernel and shrink its responsibilities. For example, the kernel being able to persist system settings is something that was possible on the Series 5 because it had battery-backed RAM available to it, so it was persisting data in the superpage and so on. This wasn't really possible in the Nokia 9210 model where a lot of that data was owned by the RTOS and persistent storage required writing to flash memory. So there were improvements in how we did the HAL [Hardware Abstraction Layer] to move from the Series 5 model, where a lot of the responsibility for persisting was with the kernel and the User Library. In the new world a lot of that either got pushed up higher into the operating system or got pushed over to the RTOS.

Those were just some of the challenges of the partner OS approach, because it was a fairly difficult piece of hardware sharing. For example, because we didn't have visibility of what was going on inside these RTOS hooks we'd get defect reports or see problems on the hardware which were kind of inexplicable. You quickly realized you needed someone in the kernel team with a big brain to think hard about the problem and discover that it would be that the RTOS was affecting the processor state in ways it shouldn't, or blocking interrupts when it shouldn't.

Other basic differences between the Nokia 9210 hardware and the original Series 5 hardware design also required some significant changes.

Morgan Henry:

In terms of architectural changes, new functionality was certainly added; things like generic support for the DMA controller and power management, as well as drawing the delineation between the kernel's responsibilities and higher level responsibilities, for example, in terms of persistence of data. Those kinds of architectural changes were happening then and those have been carried forward into the new kernel as well. So those decisions were the right ones at the time.

Another explicit goal of the EKA2 kernel architecture was to improve portability of the operating system by improving the modularity of the kernel design, so that hardware dependencies were isolated from common kernel code and so that different levels of hardware dependencies were isolated from each other, for example, to distinguish between more generic dependencies and the specific dependencies of particular devices.

The so-called 'partner operating system' solution is of course only one approach to solving the phone problem. Another goal of the new kernel architecture was to enable single-operating-system and, therefore, single-processor-core solutions.

Morgan Henry:

If you look at the problem we were trying to solve with partner OS, now with the new kernel we are in a situation where we've solved it in a better way. It certainly is a lot more architecturally sound. So the reference design team in Symbian recently announced that they'd got their first single-core-solution running using Symbian OS as the real-time OS with a personality layer. In terms of functionality, they're at the same place but with a better solution, a more robust solution.

In this approach, a 'personality' layer is used to interface the baseband stack directly to the EKA2 real-time nanokernel. The personality layer mimics the interface of the RTOS for which the particular baseband stack was written. Since the nanokernel has true real-time performance, this solution allows the baseband to be hosted on Symbian OS along with the application side, for which the extended EKA2 kernel (i.e. the nanokernel plus kernel) provides the interface, enabling a single-operating-system, single-core solution.

As well as the new kernel architecture, there have been significant other additions to the lowest levels of the system over multiple releases. A new framework for power-state management has been added to support the latest generation of phones which incorporate hardware previously found only on high-end laptops and dedicated devices such as digital cameras and camcorders, but still need to provide phone-style extended battery life.

Radio technologies, such as Bluetooth, Wi-Fi and 3G, are extremely power-hungry (power drain and battery problems were among the early technical hurdles that stalled the rollout of 3G networks). Coupled with the motors needed to drive optical zoom lenses, electronic camera flashes and large LCD displays, the power demands of high-end phones really do push the limits of battery-management technology. Simplistic three-state models (on, off and standby) which were adequate for an earlier generation of phones no longer meet the requirements. Symbian OS has been very successful at maintaining its significant lead over less-well-adapted operating systems running in mobile phones.[3]

Similarly, there have been substantial changes to keep pace with evolving generations of flash-memory technology, matching the demands of the latest phones for large removable (and non-removable) drives that provide multigigabyte internal and removable memories. File system extensions supporting flash-file storage media have been added. Flash systems are not byte-addressable, so conventional read–write mechanisms have to

[3] It has close to 80% of the market, see Chapter 2.

be completely redesigned. Flash systems also have a more limited lifetime than other memory technologies, supporting a lower, fixed number of accesses before wearing out, which means that memory accesses must be evenly distributed across all sectors of the card and not concentrated in one physical location. These all imply significant behavioral differences which must be abstracted by the file-system server to provide a common file-system interface. (The behavioral differences between NOR and NAND flash have also become significant as phone vendors have adopted the cheaper NAND technology as a way of boosting internal memory size in a cost-effective way.) Similarly, solutions such as demand paging have been explored as solutions both to RAM inflation and to mobile disk-based storage options.

Performance improvements have also been made throughout the system, at all levels from application support to the kernel and other low-level system areas. At the low level, file seek time and inter-process-communication timings have been improved, in addition to the fundamental work of managing interrupt and other low-level process latencies as part of the real-time reengineering of the kernel. In the application support layers, significant effort has been spent to maintain and improve the core PIM application engines, to keep them up to speed with current technologies as well as with evolving expectations. Thus Agenda, Contacts and Messaging have all had their share of renewal through recent releases of the operating system, particularly to improve performance.

17.5 Telephony Evolution

Telephony is an obvious area in which technology has evolved at a relentless pace since the first Symbian smartphones came to market. GSM has been enhanced with successive generations of go-faster technologies such as EDGE and with half-way house technologies such as GPRS which extend the basic voice capabilities of GSM with packet-based data services. Finally, it has evolved to full 3G, with packet-based voice and data services. 3.5G network technologies are now reaching the market and 4G will no doubt soon start to emerge (although it is not yet clear what 4G networks will mean). As well as this kind of generational network change, there is the further complication of a global market divided between competing network technologies, although GSM has, to date, dominated globally over the North American CDMA alternative. Unfortunately, 3G perpetuates the divisions. 3G evolution is based on standards but while GSM evolves to the 3GPP standard, CDMA evolves to CDMA2000 and remains as incompatible as its 2G counterpart.

Further disruptive change is promised as competing high-speed wireless technologies such as Wi-Fi and WiMAX converge with the latest