



[articles](#) [quick answers](#) [discussions](#) [features](#)
[community](#) [help](#)

Articles / Mobile Apps



[C++](#) [Windows](#) [WinMobile](#) [Visual-Studio](#) [Mobile](#)

Symbian OS design faults



Hrissan

20 Jan 2005

[CPOL](#)

17 min read

👁 269.2K

📌 40

💬 59

Rate me: ★★★★★ 4.73/5 (60 votes)

This article describes many Symbian OS design faults and searches for better decisions, unfortunately Symbian is too late to be corrected.



Introduction

The reason for this article to appear is the constant praise of Symbian OS by its developers and constant complaints of developers, who try to use this platform. It is said that a wise person admits his mistakes. But to admit the mistake, one should first understand it. The problem of Symbian OS is exceptionally bad design, in spite of 2 years of development spent only for creating the architecture.

I'll try to explain the general and specific design faults, focusing mostly on C++ development. An experience of Symbian OS programming is required to understand this article. The average knowledge of standard C++ and STL is required to understand the code examples. Also, it is required to understand, that not only mobile devices, but the developers themselves have very limited memory and processing power!

General faults (sorted from the worst to the subtle)

Complexity

According to documentation, Symbian OS is "a large OS, containing hundreds of classes and thousands of member functions". It seems to be one of the most complicated operation systems ever built. Basic Symbian OS class hierarchy contains 1201 classes. Series 60 SDK adds 293 classes and UIQ SDK adds another 705 classes! There are many classes with hundreds of methods in their interface, and less then 10 methods per class is very uncommon. No human being can fully comprehend this kind of a system.

Inability to detect things, common to different devices

Symbian OS has a common core, but the user interface part of the system interface classes are completely different. But in most places they should be identical! Let's compare the dialogs on Series 60 and on UIQ. Both have title bar with some text. Both have "control elements", listed from top to bottom. Both have associated commands (in "options menu" on Series 60 and "buttons row" on UIQ). Control elements are presented to the user to view and modify text, numbers, dates, times, etc.. They can look different and be operated by the stylus on the first platform and by soft-keys on the second one, but from the point of view of the application they behave the same. It does not matter how the command was presented and selected by the user, if it properly triggers appropriate methods. This generic approach to dialogs is best seen in J2ME. It has nothing to do with the Java language. This is just an example of good architecture against a bad one.

It is funny, that third party developers had created the set of headers, which allows the Series 60 program to compile and run on UIQ! It is very strange, that OS creators could not understand the urge of common UI structure. It could be extended (adding new control elements, commands, etc.), but it should not have to be rewritten. The problem is not only in code reusability (complete reusability is possible for most dialogs, for example), but in "developer reusability" also. It is difficult for humans to learn and understand new concepts, remember hundreds of class names and thousands of method signatures.

Making the key decision based on false facts and statements

The best example of this is exception handling in Symbian OS. The concise C++ exception mechanism was dropped and ugly "cleanup-stack" was picked, based on the claim, that "C++ exception makes the compiled code to grow 40% in size". This comparison is made in this way: the program is compiled

with or without exception support compiler option and then the size is compared. But the program, which uses exceptions, is *designed* to use exception as a safety mechanism. When exception handling is switched off, the program is smaller, but it is not safe. Many researches have found, that after another safety mechanism is added (checking for error codes, for example), not only the compiled code returns to its original size, but also the *source* code grows considerably.

After the C++ exceptions mechanism was dropped, C++ was faltered, and the other ugly things followed - two-stage construction (see Stroustrup book [^] Appendix E for explanation, why two-stage construction is bad thing), T-, R-, and C- classes with the usage restrictions, which the compiler cannot detect, prohibition of specific cases of multiple inheritance, etc. until the language stops to be C++.

Then the "experts" on the forum said: "when you start to program Symbian, you should forget everything and start from the scratch". What this really means is "we've made horrible mistakes, and you cannot write code in a clear and convenient way". If the cleanup stack is implemented in OS as the exception cleanup mechanism, then inserting `CleanupStack::Push` and `CleanupStack::Pop` is the job of the compiler, not the developer. Humans are just not reliable in such a task.

Let's see the code, generated "under hood" by the standard C++ compiler, assuming that the cleanup mechanism is implemented via cleanup stack. The classic approach to ensure proper cleanup is done in the following way: every object instance should be a member, reside in stack or be owned by exactly one `std::auto_ptr`. In the following examples, `Resource` and `Aggregate` are classes which hold some resources (for example, file or allocated memory). They both have (not inline) constructor and destructor so the compiler assumes responsibility to call the destructor and expects the constructor to throw exception.



```
class Example
{
    Resource one;
    Resource two;
    int simple_value;
    std::auto_ptr<Aggregate> aggregate;
public:
    explicit Example(int data):
        one("One"),
        two("Two"),
        simple_value( 5 ),
        aggregate( new Aggregate(data) )
    {
        // CODE1 - can throw an exception
    }
    void operation()
    {
        aggregate->operation();
    }
};
```

Under the hood, the following code would be generated by the compiler (I use C language as a portable assembler for explanation):

```

struct Example
{
    Resource one;
    Resource two;
    auto_ptr__example aggregate;
};

void example__destructor(Example * this)
{
    aggregate__destructor( &aggregate );
    resource__destructor( &two );
    resource__destructor( &one );
}

void example__constructor(Example * this, int data)
{
    resource__constructor( &this->one, "One" );
    CleanupStack::Push( &this->one, resource__destructor );

    resource__constructor( &this->two, "Two" );
    CleanupStack::Push( &this->two, resource__destructor );

    // use the knowledge that int has no destructor
    // and its constructor cannot throw an exception
    this->simple_value = 5;

    Aggregate * ag = malloc_or_throw_bad_alloc( sizeof(Aggregate) );
    // protect against exception in aggregate__constructor
    CleanupStack::Push( ag, free );
    aggregate__constructor( ag, data );
    CleanupStack::PopMemory( ag );
    // Use the knowledge, that inline auto_ptr constructor
    // cannot throw an exception
    auto_ptr__example__construct( &this->aggregate, ag );
    CleanupStack::Push( &this->aggregate, aggregate__destructor );

    // CODE1 - can throw an exception

    CleanupStack::Pop( 3 );
}

void example__operation()
{
    aggregate__operation( aggregate.ptr );
}

```

Later, if the **Example** class is used in the function, we could write:

```

void test()
{
    Example ex(1);
    // CODE2 - can throw an exception
}

```

```

    if( condition )
        return;
    // CODE3 - can throw an exception
}

```

And the compiler generates:



```

void test()
{
    Example ex;
    example__constructor( &ex, 1 );
    CleanupStack::Push( &ex, example__destructor );
    // CODE2 - can throw an exception
    if( condition )
    {
        CleanupStack::Pop( 1 );
        example__destructor( &ex, 1 );
        return;
    }
    // CODE3 - can throw an exception
    CleanupStack::Pop( 1 );
    example__destructor( &ex, 1 );
}

```

If we compare compiler-generated code to the Symbian manual-written code, which does the same thing, we can see that the generated code looks very similar to the code, which every Symbian developer writes everyday, but it is sometimes faster (!), as part of the objects can be effectively stored in stack or made class members, instead of allocating them in heap. Source code for standard C++ is short and simple, and it is guaranteed to cleanup everything! Note the comments in the generated code to see how the compiler intelligently uses its knowledge of destructor and constructor code (if they are **inline** or generated automatically) to decide whether certain code should be generated. T, R, C classes are not required any more! Modern compilers are wise enough to insert cleanup mechanism only to places where it is actually required.

Note, that the correct cleanup does not depend on how the exceptions themselves are implemented. If arbitrary classes cannot be thrown due to some limitations (this requires a form of dynamic type identification), then **throw/catch** argument could be restricted to use **int** or the special exception class or a predefined set of classes. Sacrificing exception types is bad, but it is tolerable, because humans can still easily use them, while sacrificing automatic cleanup is not tolerable, because the humans will have to face the task, which he or she is not designed to solve effectively.

Lack of documentation and clear examples

This is just the consequence of the complexity. But things could be better, if the examples were the source code of real programs. The program, which demonstrates hundred types of lists, is not a programming example. It looks pretty well for the non-technical person (probably a manager in

Symbian), who can run and play it. But the developer, who looks for information in source code, finds nothing helpful.

Static data

The documentation explains lack of static data in this way: "Implementation is difficult, as DLL could reside in ROM, and so we do not have the place to store static data. Besides, the global variables are bad for OOP". While the global variables are considered a bad thing in OOP, the class static variables (and module static variables) are not. In fact, class static variables are very common thing in OOP to store the information common for class instances. The implementation impossibility is also a false claim, but to understand it, let's consider the following example, written in standard C++.

```
class Example
{
    static int state;
};
```

And then (when the state is required).

```
int s = Example::state;
```

Under the hood, the compiler adds static data to data segment. It looks, like this:

```
struct DataSegment
{
    ...
    int example_state;
};
```

And later, to access it, the compiler generates:

```
int s = get_data_segment()->example_state;
```

where `get_data_segment` is an imaginary function, returning the address of the data segment (on most architectures it is stored in processor-specific register). Of course, Symbian application has to store its data somewhere! In Symbian, all the program data is stored in the document.

```
class MyDocument : public CAknDocument
{
```

```
...
int example_state;
};
```

And then (when the state is required)



```
int s = static_cast<MyDocument *>(
    CEikonEnv::Static()->EikAppUi()->Document() )->example_state;
```

It is surprising, but from a low-level point of view, there is very little difference in these two approaches! As traditional OSs keep track of data segment, Symbian OS application manager keeps track of document. Placing all the static data in the Document manually, the Symbian developer is again making the job of the compiler tougher! But this time the developer is in a worse situation. Consider the following code:



```
template<class T>
class RequiresStatic
{
    typedef T state_type;
    ...
    static state_type state;
};
```

In standard C++ for every used instantiation of `RequiresStatic`, the compiler adds its state to the data segment.



```
struct DataSegment
{
    ...
    RequiresStatic<int><int />::state_type required_static_int_state;
    RequiresStatic<double>::state_type required_static_double_state;
    RequiresStatic<Color>::state_type required_static_Color_state;
};
```

But the Symbian developer is unable to do the same thing and add these variables to `MyDocument`! The developer cannot know which classes were instantiated, especially if this template was in the class library. Anyway, doing the job of the compiler is something very error-prone for humans, especially if this requires writing a lot of code. It is also funny to note, that the encapsulation of class and module static data is breached, as they can be accessed by anyone who has access to the document. Symbian attempt to enforce object orientation breaks it!

Strange (not technical) design goals

Symbian developers were creating an "Object oriented" system, rather than a useful one. "Object oriented" is not a synonym for "Good". OOP is just a method, which the developer is required to master to build a system which is easier to devise, code, document and (later) understand. Symbian OS complexity is apparently a good indication of wrong application of object oriented approach.

C++ Standard Library and STL

STL is basically the library of generic containers and algorithms with iterators gluing them together. Symbian developers decided to implement their own library of containers (they completely ignored algorithms and most iterators, though). The documentation informs that the reason for non standard implementation is efficiency. Let's consider the Symbian implementation of strings, arrays and lists.

Strings in Symbian are called "descriptors". They are templates and each descriptor can hold a sequence of 8 or 16 bit characters. There are basically two groups of descriptors - "modifiable" and "non-modifiable", the former is inherited from the latter. Strange enough, non-modifiable descriptor allows complete replacement of content (similar to pointer semantic of `char *` type). Value semantic of `std::string` uses the usual `const` keyword to mark non-modifiable objects. I could not find any situation where the pointer semantic of non-modifiable descriptor can increase the effectiveness of the code. Keeping value semantic would require two times less classes.

In each group, different flavors of descriptors exist. The base descriptor for modifiable descriptors is `TDes` and the derived descriptors are `TBuf`, `TPtr`, `HBuf`. The `virtual` methods are not used, instead, the base descriptor stores a derived class type in bit field and uses table method invocation. This saves memory, but slows down execution, compared to `virtual` functions.

The type `HBuf` (heap descriptor) has approximately the same operations, memory and time requirements as `std::string`. `TPtr` stores the pointer to external data and `TBuf` stores the data inside and is ideal for creating short strings in the stack. They both are special and useful classes, required for efficiency. The best decision would be renaming `HBuf` to `std::string` and making all descriptors to have the interface of `std::string`. So, at least part of the code could be reused between platforms and the developer could reuse his knowledge and experience with `std::string`. It is worth saying, that `std::string` lacks a very useful ability – to "lock" the string and return non-const pointer to the data (Symbian descriptors have this ability). Adding this method to `std::string` would make certain code not portable from Symbian OS, but this is a small change and it is easy to remember. And it is required mostly in low-level code, which calls OS functions and thus is not portable anyway.

There are a multitude of array and buffer classes in Symbian. Some arrays use single flat memory chunk and others use several segmented chunks. They differ in effectiveness of different operations the same way as `std::vector` and `std::list`. There is also a circular buffer, which is very similar to `std::deque`. There are also special array classes for storing descriptors and pointers. Also, Symbian has the classes to implement single and double linked lists. No Symbian array or list gives any speed or memory advantages compared to STL containers, with the exception of packed array. (STL often uses specific optimizations, for example `std::vector` can use `memmove` instead of loop with `operator=` to reallocate simple objects. Such optimizations are almost always based on type traits mechanism.) This array stores variable-sized objects in flat memory chunk. Such an array does not fit to STL well (it could

not be sorted effectively with `std::sort`, for example. But as `std::list` provides its own `sort` method, then packed array would be able to do the same.). So the best decision would be to support STL containers and add a packed array class, if it is required.

If `std::map` is considered to be too complicated for mobile devices, then it is worth pondering, that if it is really required, then the developer would manually implement it. The application developer should make the decision to use it or not, not the creators of the platform. The platform creator should just provide the application developer with good tools.

Document-View architecture

Symbian OS boasts of Document-View architecture with `Application`, `Application UI`, `Document` and `View` classes mandatory for every application. We've seen that document is required instead of data segment (`static` variables). Also it provides an interface to the application so that the OS can call its methods. But what are the `Application`, `Application UI` and `View` classes? It is clear, that single `View` is not part of every application - there are applications, which only have a dialog. Or applications, which run in the background and communicate using "Alerts" (message windows). Also there are many applications, which have several views. Thus making every application to have one `View` is a wrong thing. `Application` class in Symbian is the class, which constructs the document and (again) provides interface to the application. What is `Application UI`? `Application UI` is (another) class that provides interfaces to an application, specifically interface for handling commands. When several things do the same, it is wrong (and this also can be funny). For example, the method `Application::Save` saves a document and the method `Document::Save` is called to ask a program to attempt to free the occupied memory. As the only thing required is an interface to the application, it would be much better to name it `Application` and throw the other classes away. If a particular developer decides to use the Document-View architecture in his particular application, let him.

Resources

Using resources is in general a dangerous thing, because it separates the code into several files and several programming languages. The system should provide the way to bind the C++ code and resource definitions and this is not an easy thing! Resources are useful, when you can change them without changing the executable. But how can you really accomplish this? If you add the button to the resource, how do you add behavior? It is impossible. Then editing resources become very limiting - you can add static labels and you can define constants (like translating text to the other language or modifying the slider maximum value). But as you cannot define the parts of a resource, you should copy all its structure and using this feature for translation becomes costly. Later, when the structure changes (splitting dialog in two tabs, for example), you should carefully repeat this on every translation. The inherent problem of resource is that you should accurately match the code with the resource. Look, the J2ME program (which creates all controls in the code) is several times shorter, than the Symbian C++ program plus its resources. And believe me, defining unique constants for control identifiers is just another job of the compiler, not of the developer. By the way, resource definition language is another language, which we should learn (and it is far from perfect).

TBool, TAny

TBool is a tiny fault. If the compiler supports **bool**, this is not required. If the compiler does not have built-in **bool** type, it is better to define **bool**, **false**, **true** and recommend upgrading the compiler as soon as possible. (Such a prehistoric compiler definitely has issues with templates, exception handling, etc.)

TAny is another tiny fault, it adds nothing new to the C++, it is just a synonym for **void**. Small things, but every developer should understand this and store this in his or her brain. If something is not required, throw that away! Everyday the thought of a developer should be "What can I throw away from the system?", not "What can I add to the system?". **TAny** is an obvious example of the latter approach.

Faults specific to Series 60 (most of them are corrected in UIQ)

Error handling in emulator

Most errors are detected by the special code. When they are detected, they should be reported as accurately as possible. For example, the program can try to remove the menu element, which does not exist in this menu. It is apparent (for anyone besides Nokia), that the message should appear "Unable to find menu element 1013 in method **Menu::remove**". Breaking into the debugger is also required in debug mode (especially for more generic errors, like "array subscription index out of bound"). But on the Series 60 emulator, such programs crash immediately with the informative message - "Program closed, OK". Almost any mistake leads to a similar crash with this notorious message.

Things do not work

For example, when you try to display an alert during the construction of a View, the alert does not appear without any error or explanation. Superstitious? No, just another consequence of complexity. Such problems are very common in Series 60 world. Have you ever read the forum Nokia? Hundreds of people keep asking the same things, like "How can I display the list?", "How can I add menu to ...", "In dialog ... does not work", and so on. People cannot do these simple things because these things just do not work, not because the people are stupid.

Two Exit buttons

It is very funny, but (according to documentation) Nokia adds an "Exit" command to the main menu of every Java program (I doubt this can classify as breaking the standard, which of course does not say "the system should not add random commands to the application menu"). Nokia understands, that this breaks portability (Documentation states that "The developer should consider between portability

and two exit commands"). Moreover, it breaks portability deliberately. It is clear for any programmer, that before displaying the menu the system can look for the **Command.EXIT** command in the menu. It does not.

Conclusion

Despite giant efforts spent to build such an enormous system as Symbian (with UI variants, like Series 60 or UIQ), it is clear, that the result is far from perfect. Symbian OS should be considerably redesigned to meet its design goals, especially "provide users with a richer mobile experience". Nowadays most of the developers spend their time not on application-specific tasks, but to fight the bad system design. While Symbian makes humans to fulfill several tasks of the compiler, the fight is definitely lost. Bugs, impossible in most modern systems, live comfortably in Symbian applications. But Symbian creators do not want to admit their mistakes and continue to insist that Symbian is very good.

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

Written By

Hrissan

Web Developer

 Russian Federation

This member has not yet provided a Biography. Assume it's interesting and varied, and probably something to do with programming.

Comments and Discussions

You must [Sign in](#) to use this message board.

Search Comments



First Prev Next

 Old article, but new... 	 cuigri	20-Nov-08 4:44
 bluetooth connectivity of symbian os 	 ravilikesaboli	27-Mar-08 20:01
 Re: bluetooth chatting of symbian os 	 ravilikesaboli	23-Apr-08 2:09
 to put an opinion into the basket [modified] 	 cprojectaccount	21-Mar-08 5:00
 I do not agree 	 mrtj	29-Feb-08 3:40
 I agree completely 	 jdekeij	5-Nov-07 3:23
 TDes 	 ZogOBog	20-Aug-07 15:07
 STL on symbian available 	 Vince Ricci	15-May-07 22:56
 Re: STL on symbian available 	 Vince Ricci	15-May-07 23:36
 I completely agree 	 TemplMetaProg	13-Jan-07 9:37
 writting a calculatore with c++ which run on a nokia 	 zahraonline	10-Nov-06 21:17
 Re: writting a calculatore with c++ which run on a nokia 	 Saeedqwe	30-Dec-06 21:27
 one confusiton about cleanup stack 	 ankitapatel	14-Apr-06 17:41
 Solution? 	 Ilya Minkov	1-Jan-06 9:34
 Nice article 	 Anonymous	11-Oct-05 9:37
 Re: Nice article 	 John Vaudin	6-Oct-06 2:45
 help me ! explore builder c++ / phone uiq 	 passager51	24-Jun-05 5:06
 Help Me!!! How to write startup code for mobile 	 Girishmk	27-Apr-05 8:23
 about writeable static data 	 Anonymous	25-Apr-05 2:29
 a lie about writeable static data 	 Anonymous	25-Apr-05 2:27
 THANK YOU!!!!!! 	 Anonymous	10-Mar-05 2:30
 Simplicity and elegance is rarely understood 	 dfdzg	24-Feb-05 4:33

 Symbian OS and ARM CPU Target 	 AKSIVAKUMAR	9-Feb-05 21:01
 Yes but Symbian v9 is coming.... 	 JjZack	4-Feb-05 12:23
 Re: Yes but Symbian v9 is coming.... 	 TomM	8-Mar-05 12:01

Last Visit: 31-Dec-99 18:00 Last Update: 23-Aug-24 6:04

[Refresh](#)[1](#) [2](#) [3](#) [Next](#) 

 [General](#)  [News](#)  [Suggestion](#)  [Question](#)  [Bug](#)  [Answer](#)  [Joke](#)  [Praise](#)  [Rant](#)  [Admin](#)

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

[Permalink](#)
[Advertise](#)
[Privacy](#)
[Cookies](#)
[Terms of Use](#)

Layout: [fixed](#) | [fluid](#)

Posted 20 Jan 2005

Article Copyright 2005 by Hrissan
Everything else Copyright ©
[CodeProject](#), 1999-2024

Web03 2.8:2024-07-22:1