

Runtime Environments – the Concept

Roy Ben Hayun

Published by the Symbian Developer Network

Version: 1.0 – March 2007

| | | |
|----------|--|----------|
| 1 | INTRODUCTION | 2 |
| 2 | REQUIREMENTS OF RUNTIME ENVIRONMENTS..... | 2 |
| | 2.1 CLOSE RELATIONSHIP TO OTHER VARIANTS OF THE SAME RUNTIME ENVIRONMENT | 2 |
| | 2.2 PERFORMANCE OF THE RUNTIME ENVIRONMENT ENGINE..... | 2 |
| | 2.3 USER AWARENESS AND EXPERIENCE | 3 |
| | 2.4 INCREASING A DEVELOPER’S PRODUCTIVITY | 3 |
| 3 | MAIN ELEMENTS OF RUNTIME ENVIRONMENTS..... | 3 |
| | 3.1 COMPUTING LANGUAGES..... | 3 |
| | 3.2 LIBRARIES..... | 6 |
| 4 | INTEGRATION WITH THE UNDERLYING SYSTEM | 8 |
| 5 | RUNTIME ENVIRONMENTS GO WITH TRENDS..... | 8 |
| 6 | NEXT TIME..... | 9 |
| 7 | GLOSSARY..... | 9 |

1 Introduction

This is the second instalment of Mobile Runtime Environments, the column that gives developers, technical leaders and technical decision-makers an insight into the various development platforms available for Symbian OS.

A Runtime Environment can be defined as an execution platform that is hosted by, and receives services from, the underlying native operating system. Well known examples of mobile Runtime Environments are Java, Flash, Ruby and Python.

The focus of this month's article is the Runtime Environments concept – what is a Runtime Environment and what are the different parts of a Runtime Environment?

To answer those questions, we first explore the requirements of a Runtime Environment. Secondly, we investigate the main components of a Runtime Environment such as computing language and libraries. Next, we discuss the relation with the hosting native platform. Finally, we try to define criteria for successful Runtime Environments.

2 Requirements of Runtime Environments

2.1 Close relationship to other variants of the same Runtime Environment

A mobile Runtime Environment usually evolves from a superset of the equivalent Runtime Environment hosted for another platform. This superset could be very different in its usage; for example, Flash-Lite for mobile devices evolved from Flash that runs on desktops and JavaME evolved from desktop Java technologies.

During this evolution process, some changes may have been applied to the language, libraries and frameworks of the originating Runtime Environment, and therefore the two platforms, whilst sharing a lot in common and seen as the same Runtime Environment, are no longer fully compatible. So, for example, a Python script that runs on a PC will need some modification in certain areas before it will run on a mobile device.

The changes to the language or library are inevitable and essential for the porting of the Runtime Environment to a different platform. However, the Runtime Environment still needs to be a well defined subset of the original standard to ensure effective compatibility and portability. Unless this is done, the mobile Runtime Environment will be a remote offshoot rather than a platform in which developers can apply the same skills and knowledge to both. For example, a JavaSE developer will need to adjust to JavaME, but will feel that this is a similar Java Runtime Environment, and the learning curve will be minor compared to moving to a completely different language.

2.2 Performance of the Runtime Environment engine

A non-native Runtime Environment is a software program which is hosted on the native platform. It could contain an interpreter, JIT (Just In Time) compiler, libraries, interfaces to the system applications management and much more. The Runtime Environment, which is a considerable sizeable application, must not affect the performance of the device or the performance of the application. For example, the Runtime Environment engine should not choke the CPU or grab resources exclusively for the hosted applications and should be as efficient as possible in terms of dynamic and static resources.

2.3 User awareness and experience

When running a non-native application with a Runtime Environment, the user should not be aware that the application is hosted and is not running directly on the OS.

Lifecycle activities like installation and launching should be easy to use and the user should follow the same actions needed for native applications. Having separate installation and launching mechanisms will require the user to perform actions which are not the same as they're used to.

Installation and uninstallation of a Runtime Environment hosted application should work in the same way as installing and uninstalling a native application.

When launching the application, the separate launching of the Runtime Environment and of the application itself should be transparent and seamless to the user.

During the runtime of the application, it should have a similar look-and-feel as the native platform, and the same user interaction model as native applications.

2.4 Increasing a developer's productivity

A Runtime Environment needs to increase productivity to help reduce both Time-To-Market and development costs.

Productivity is still a problem in computing and is currently becoming an increasingly important factor in software engineering. There are many processes that aim to help but are not specific to any computing language or Runtime Environment.

The Runtime Environment could contribute to the cost reduction by having the right tools, and by having a language design which can be written correctly, quickly and easily.

For example, Java has state of the art tools for all development stages and the language itself was designed for rapid and robust development.

3 Main elements of Runtime Environments

In this section we will take a deeper look into what a Runtime is. We will discuss the key elements – the language, the libraries and the interaction with the hosting platform.

3.1 Computing Languages

The computing language is the medium by which we express algorithms and computations, and instruct machines what to do. Thousands of programming languages have been created and evolved over time and have been designed for many different types of tasks and systems. Although there have been discussions on how to create a single universal programming language, at the moment it is the common belief that you need different tools for different tasks and therefore there are different languages for different usages.

3.1.1 Programming languages evolution

Tracking the evolution of languages gives a lot of information on where language features came from and how different features can be used in conjunction. So before looking at some of the features of computing languages, let's go through a quick overview of the history of computing languages in the last 50 years.

The first high level programming languages were developed in the 1950s. Some disappeared immediately, some barely managed to survive and some have evolved and even prevailed. By the

mid 50s, assembly languages were invented and assemblers produced the machine code from mnemonics. During the second half of the 1950s came the first high level language, Fortran, closely followed by Lisp, Cobol and Basic.

The 1960s introduced Object Oriented Programming (OOP) languages; the first was Simula, which introduced objects, classes, subclasses, garbage collection, etc. During the 1970s the more notable languages were Smalltalk which added metaclassing, C language for low level power, Ada for defence systems and the logic-programming language Prolog.

The Object Oriented paradigm became prominent in the 1980s and went mainstream in the 1990s with the rise of C++.

More recent innovations like mark-up languages, scripting languages, Java and the .NET platform bring us to the world of software engineering as we know it today.

3.1.2 Computing language features

When designing a Runtime Environment, the designers need to make numerous decisions on the language features which will be used or would be left out. Those decisions are heavily influenced by the designers' philosophy of computing languages, their previous experience and the requirements which the Runtime Environment needs to satisfy.

There are thousands of computing languages which vary in their features and philosophy, and we will try to look at those features from the eyes of a language designer. Although it would be interesting, we can not go through the whole list so we will focus on a few main features: Execution Models, programming paradigms, the Object Orientation paradigm and typing systems.

The Runtime Environment Execution Model defines how the program is executed on the device, and hence also the build and development process.

- A native Execution Model is where that the program is compiled into object code and executed directly on hardware like C++.
- One interpreted Execution Model specifies that the program instructions are purely interpreted, like Ruby on Symbian OS.
- Another interpreted Execution Model compiles the source into intermediate code which is then interpreted on the device.
- A more advanced interpreted Execution Model may be where the intermediate code is compiled on the device prior to the execution or on the fly by a JIT compiler, as in current Java technology.

A programming paradigm is the symbolic view that the programmer has of the logical execution of the program (this is as against the actual low level execution).

First we could distinguish between Descriptive Languages, that describe what things should look like (i.e. HTML, SVG and other mark-up languages) and Imperative Programming, in which computations are carried as a series of command statements that change the program state, like Java, Python or C++ etc. Imperative Programming contains many programming paradigms such as Object Orientation, Procedural, Generic etc.

A language can support a single programming paradigm or it can support multiple paradigms. For example, Java supports a single Object-Oriented paradigm while C++ is a multi-paradigm language which supports procedural, generic and Object-Oriented programming. The mechanism of supporting multiple paradigms may also change; for example Ruby is primarily an Object-Oriented language yet supports the procedural paradigm in an Object-Oriented way by dynamically adding global functions to the base Object class.

Procedural Programming breaks the tasks into data structures and routines which contain a series of instructions and calling of other functions to be carried out.

Generic programming is a way of specifying variable parts which later can be instantiated by specifying the concrete instances or types that fit into those variable parts. This extended language syntax enables generic variable type declarations, generic classes subtyping, generic methods, generic containers and generic algorithms.

Let us spend a bit more time on the Object Oriented paradigm, examining how different variations in the OOP principles are applied on various languages. By way of an example, the inheritance model may be either single inheritance or multiple inheritance where a class inherits from more than one base class. Other inheritance idioms are found in Java interfaces, Ruby modules or C++ pure virtual classes. Inheritance can be class-based, where Objects are constructed according to a class template, or it can be prototype-based where new objects are cloned from an existing instance.

If you ever experimented with metaclassing, then you have seen how there could also be multiple levels of classes and objects. Starting from single hierarchy systems where objects and classes are viewed similarly, there is then a higher level where objects are instances of classes which are not accessible, and in the highest levels of metaprogramming there are three distinct kinds of objects - objects, classes, and metaclasses.

The language implementation of the Encapsulation principle, which hides the implementation details from client objects, can vary. For example, Java adds the “protected” keyword and C++ has the notion of “friends”.

Another interesting example of a different Object Oriented flavour is the level of Object Orientation purity. A language can be pure or hybrid Object Oriented. Java is an example of a hybrid OO language in which primitives like *int* and *short* are not objects, for efficiency reasons. On the other hand, Ruby is a pure Object Oriented language in which everything is an object, including numbers.

Going back to the main language features, let us discuss the typing systems which define how the programming language classifies values and expressions into types.

Most of the languages currently used can be divided into two categories: dynamic or static.

A static typing system is when a compiler checks that the type used is compatible with the declared type. For example, Java is a statically typed language in which the type check is performed at build time.

In dynamic typing systems, such as in Ruby, the type check is deferred to runtime and only then is the type used checked to ensure it is compatible with the type declared. Typically, dynamically typed languages are considered to be saving programming time and the software is more flexible and amenable to change. However, type errors cannot be automatically detected and the code must be actually executed to ensure it is correct.

3.1.3 So many language features... what does it all mean?

There is a wealth of features available to languages which enrich the language including:

- Block – A block of code may be passed to a method, which could invoke that block using a construct like Ruby’s `yield` statement.
- Closure – This pairs the block with the environment and when called, the block can use the environment context.
- Continuation – Some languages allow the execution state to be encapsulated in an object and later restore the state from this object.
- Regular expressions - A string using certain syntax rules, which describes patterns of sets of strings.
- Threading – Can be a core language element or part of the Runtime Environment libraries.

Additional features in that list would include Lambda expressions, error handling, exceptions, memory management, uniform access, reflection, access control, synchronization etc.

As discussed earlier, the designers of a programming language need to make a set of decisions on what is included and what is scoped out of the language, but it is probably true to say that no language can be perfect for every person or for every task. There are two things to be considered—what is it that is common to Turing-complete languages and what is it that is out of the scope of the computing languages?

Firstly, everything that one Turing-complete computing language can compute can also be done by other Turing-complete languages, but the cost and effort will vary depending on the design and features. So if there are two features which enable the same thing, it is more powerful to provide both of them but it makes for a simpler language to pick just one of them and support this one alone. The trade-off will be in the code readability, execution speed and language learning curve.

Secondly, the computing language gives computational ability, but other capabilities, which are required for most of today's software, are not in the scope of the language. This brings us to the next main element in a Runtime Environment— the libraries.

3.2 Libraries

In the previous section we discussed computing languages, which deliver the computational capability of the Runtime Environment. However, if you want to connect over Bluetooth, display a UI or use persistent data, then this is all done by using the libraries and not by the computing language. So now we will examine the Runtime libraries which bring the Runtime power to most of today's applications.

First we will define the relation between standard and optional libraries. Secondly, we will discuss how libraries give support for technologies. Finally, we will discuss the ability to dynamically extend the Runtime with libraries.

3.2.1 Standard and optional libraries in different market segments

A Runtime Environment stack could be composed of several libraries, of which some are standard and some are optional, depending on market segmentation.

Most programming languages have an associated standard library, which is conventionally made available by all implementations of the language. This core library typically includes definitions for commonly used algorithms, data structures, and mechanisms for input and output.

The standard library is what every platform supplier must provide in their implementation of the Runtime Environment.

An optional library is a set of APIs that is layered on top of a defined stack but is not mandated. It allows the definition of APIs that can be added flexibly and can be independent of particular functionalities. Optional libraries typically include APIs that are far more domain-specific than the standard libraries.

This separation into what is part of an optional library and what is part of a standard library could change when looking at different market segments. An API which is defined as an optional library in one vertical segment, could be a part of the standard library in another niche market to guarantee interoperability within a vertical device family or domain.

Optionally, a Runtime Environment can provide a mechanism to dynamically add new libraries to be used across all applications (i.e. adding shared DLLs or Jar files to the JavaSE CLASSPATH). The alternative is when the Runtime Environment is not extensible and the supplied set of libraries

is the only set accessible across different applications. For example, unlike JavaSE, there is no possibility to add Jar files in JavaME to be shared by many applications.

3.2.2 Technologies coverage by libraries

In the age of converged services, applications use increasingly different technologies and so we will look at how to evaluate Runtime libraries' support for technologies. There are three fundamental steps in this evaluation – one, does the Runtime have any support; two, what is the width of the library support; and three, what is the depth of the library support?

The first question gives the simple yes/no answer to “does a Runtime have libraries that support a certain technological area?”

Some technology areas are more crucial than others for a Runtime Environment. For example, not having any UI library begs the question of whether that Runtime will survive at all and what purpose it serves on a mobile device.

A less obvious case is library support for a specific service, the availability of such support determining if the Runtime Environment can be used as a platform to provide users with that service. For example, if we want to create an IMS (IP Multimedia Subsystem) application using Python, we could ask “does Python support SIP?” Or if we want to display rich content we could ask “does JavaME support mobile multimedia?”

In other technology areas, a library support can be seen as a ‘nice to have’ but not a ‘must’ requirement or it could be replaced by some other mechanism. For example, DBMS support may be replaced with file access or some other persistence mechanism like JavaME RMS (Record Management System). It might be that the Runtime Environment power would seem to be less appealing but the lack of that capability can be overcome.

The second question looks into a specific technology domain to enumerate which features of that technology the Runtime library supports.

For example, having confirmed that there is support for multimedia, the second question allows you to enumerate which multimedia capabilities a Runtime has. This would answer more specific questions, such as whether the Runtime has Camera support that enables you to take a snapshot or whether it enables you to play audio and video.

Another example from another technology area is: if the Runtime has library support for remote connectivity, then which protocols does it support, and does it for example provide support for protocols like TCP and UDP?

The last question answers specific needs, which are not always common use-cases, and relates to the fine grained control of the library in that technology area. Naturally, the library APIs will expose a finite control over an entire technology and most applications would use the most common use-cases. Generally the more fine grained the control, the more powerful is the Runtime support.

If we follow the previous examples, answering the first question indicated that there is a library which provides mobile multimedia support and the second question indicated that there is support for taking snapshots with the inbuilt camera. The third question will reveal what is the parameterisation for those common uses-cases and is it possible to have fine grained control like using red eye reduction filtering when taking a picture or to set the number of video frame rates.

In the example of remote connectivity, if there is remote connectivity, and there is TCP support, a fine grained control could be to set the Socket TCP parameters.

An important point to keep in mind is that a Runtime Environment is only as powerful as the underlying host platform. The more powerful the hosting platform, the more powerful the Runtime Environment, which can expose those rich services with its high level APIs. Symbian OS is a very

rich platform which provides a wealth of services and frameworks such as connectivity, security, protocols, graphics, messaging, multimedia, networking, telephony and more. This large range stretches the potential of every Runtime Environment implementation which is hosted on Symbian OS and enables it to be a competitive platform.

4 Integration with the underlying system

The host platform has idioms and features which a Runtime Environment may expose to a programmer or insulate them from, depending on its design.

First, the language may enable a mechanism to execute code written in the native language of the host platform, for reasons like performance boost or interaction with low level modules. It is very common for languages that were created after C was introduced to provide integration with C. Such mechanisms enable high level languages to provide access to the low level power that make C great for certain purposes while keeping the high level language clean of those constructs. For example, JavaSE, Ruby and Python have interfaces to C, which can also be used as a springboard to C++ code. On the other hand, in CLDC/MIDP, those Java interfaces to C were removed and the application can execute inside the JavaME sandbox only.

Secondly, the language or libraries could expose platform specific idioms and provide a less insulated environment in which the programmer is aware of the particular hosting platform. Python for Symbian OS minimizes the exposure to the native Symbian OS Active Objects idiom but certain Active Object-based services are still available in the e32 module.

Finally, security is a major concern on every platform and so Symbian OS v9 introduced the concepts of Platform Security, capabilities and the mandatory Symbian Signed process. However, a Runtime Environment may have its own security policy (i.e. MIDP security policy); or it could reuse Symbian OS Platform Security (i.e. Python or Flash-Lite passive content).

5 Runtime Environments go with trends

The last section in this article will discuss how the Runtime Environment evolves or changes to support market and software engineering trends. The dynamic nature of a Runtime Environment will ensure it remains valid and applicable as a development platform for the longer term, something that is important to evaluate before investing budget and effort for development on that platform.

The risk of a Runtime Environment which does not evolve constantly is stagnation, which will confine it to the market segment which it serves currently and eventually will cause other development platforms to take its place in those areas as well. Subsequently, an application hosted on such a Runtime Environment will become obsolete and would have to be re-implemented on another platform. On the other hand, a dynamic Runtime Environment, which provides solutions to new business and technical requirements, will maintain its market share and possibly expand to other existing and new market segments. Software written for that Runtime Environment will remain viable.

Some quantification is required in order to mitigate the risk of software becoming obsolete and evaluate how the Runtime Environment evolves.

First, we need to identify the technical and business trends associated or affecting the domain of a specific Runtime Environment and quantify how powerful those trends are. An example for a trend that affects a Runtime Environment is Open-Source projects in which the VM code is becoming publicly available and allowing more innovations to be integrated more quickly into the platform. Other examples are IMS, which adds a large number of requirements to the list of services that

should be implemented with a Runtime Environment, and Web2.0, which is putting further focus on increasing productivity - still an open problem in software engineering. One of the characteristics of Web2.0 is the use of high level languages for development.

We then need to identify the progress indicators which show if the Runtime Environment is going with the trends and continually evolving. Such progress indicators could be library support for IMS services, expansion to other deployment platforms or market segments, and the Runtime being included in technical roadmaps of market leaders. In the case of increasing productivity, progress indicators are the increasing number and innovations in development tools for that platform and if the language design is being re-examined to ensure it is matching the changing requirements.

6 Next time

In the next instalment we will examine the phenomena of mobile Runtime Environments and evaluate each of them. We will look at major Runtime Environments like Java and C++, niche Runtimes like Flash and the popular scripting languages Ruby and Python. For each we will review the history, computing language, libraries, advantages and shortcomings.

7 Glossary

The following technical terms and abbreviations are used within this document.

| Term | Definition | Reference |
|------------|-----------------------------------|---|
| Flash | | http://www.adobe.com/products |
| Flash Lite | | http://www.adobe.com/products/flashlite/ |
| IMS | IP Multimedia Subsystem | http://www.3gpp.org/specs/numbering.htm |
| JavaEE | Java Enterprise Edition | http://java.sun.com/javaee/ |
| JavaME | Java Mobile Edition | http://java.sun.com/javame/ |
| JIT | Just In Time | http://en.wikipedia.org/wiki/Just-in-time_compilation |
| MIDP2.0 | Mobile Information Device Profile | http://java.sun.com/products/midp/ |
| OPL | | http://developer.symbian.com/main/tools/opensrc/languages |
| Perl | | http://developer.symbian.com/main/tools/opensrc/languages |
| PJava | Personal Java | http://java.sun.com/products/personaljava/ |
| Python | | http://developer.symbian.com/main/tools/opensrc/languages |
| RMS | Record Management System | http://java.sun.com/javame/reference/apis.jsp#api |
| Ruby | | http://developer.symbian.com/main/tools/opensrc/languages |

| Term | Definition | Reference |
|-----------------|-----------------|---|
| Turing Complete | | http://c2.com/cgi/wiki?TuringComplete |
| VM | Virtual Machine | http://en.wikipedia.org/wiki/Virtual_machine |
| Web2.0 | | http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html |

[Back to Developer Library](#)

Want to be kept informed of new articles being made available on the Symbian Developer Network?

[Subscribe to the Symbian Community Newsletter.](#)

The Symbian Community Newsletter brings you, every month, the latest news and resources for Symbian OS.