On the Structure of a Software Product-Line for Mobile Software

Tommi Myllymäki, Kai Koskimies, and Tommi Mikkonen

Institute of Software Systems, Tampere University of Technology
Box 553, FIN-33101 Tampere, Finland
{tommikm, kk, tjm}@cs.tut.fi

Abstract

A software product-line is a set of software systems sharing the same architecture, implementation platform, and application development facilities. In this paper, we begin by introducing a general model used in multiple software product-line systems in different domains. Based on the model, we show how the Symbian architecture reflects the different parts of the model in the domain of mobile system software. This model motivates different product configurations as well as flavors of product variance. Moreover, the model also determines the design concerns that are to be advocated in the design of individual components belonging to the product-line.

Keywords: *Software product-lines, product-line architecture, Symbian OS*

1 Introduction

A software product-line is a set of systems that share a common architecture and a set of reusable components. Jointly, they constitute a software platform supporting application development for a particular domain [1], [2].

As already analyzed in [4], a general model for architectures in software product-line systems imposes the dependance of software components on architectural aspects at different levels of abstraction and generality. The levels referred to above can be characterized as follows. To begin with, a software component may depend on the form and semantics of some general resources common to a wide spectrum of products. Secondly, a software component may depend on a particular, general architectural style. Thirdly, a software component may depend on the architectural decisions characteristic to a particular application domain or type of a product. Finally, a software component may depend on the design decisions required only for a particular product. The same structure has emerged in several different environments, ranging from networking infrastructure to Java-based systems and mobile devices, as discussed in [4].

On the basis of these dependencies, we divide the components of a software system into four categories, each constituting a layer. The layers are called the *resource platform*, the *architecture platform*, the *product platform*, and the *product* layer, respectively. The resulting layered architecture is depicted in Fig. 1.

The layers can be ordered on the basis of generality. Above, the categories were listed in the order of decreasing level of generality. If a software component depends only on the way general resources are provided but not on any particular architectural style, it belongs to the resource platform layer (i.e., the lowest layer in the figure). If a software component depends on a chosen architectural style, but not on a particular domain or product category, it belongs

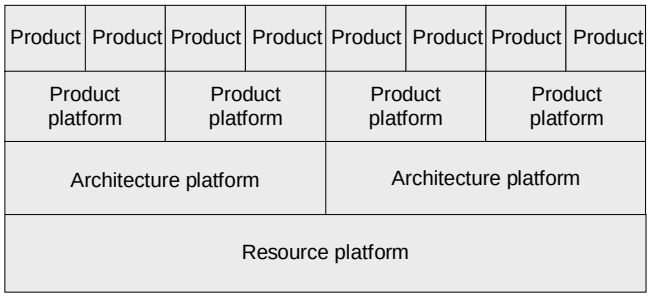

Fig. 1. Layered platform architecture.

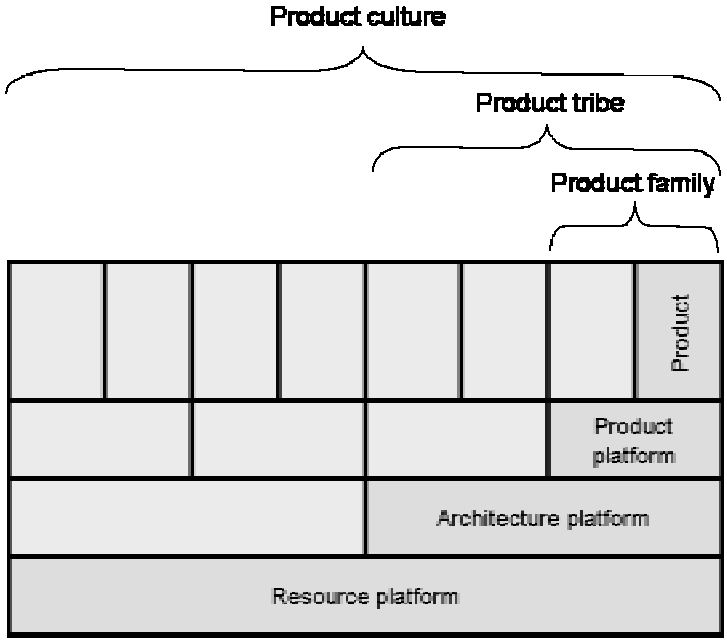

Fig. 2. Product hierarchy. The parts required for a single product are shaded.

to the architecture platform layer. If a software component depends on an application domain or on the type of a product but not on a particular product, it belongs to the product platform layer. Finally, if a software component depends on a particular product, it belongs to the product layer (uppermost layer). All the software artifacts belonging to any of these layers are collectively called a product-line.

The products built on a software product-line can be divided into a system hierarchy based on the level of shared layers. Products that share the same product platform are called a *product family*. These products have a significant amount of common functionality, and they share the same architecture. Products that share the same architecture platform are called a *product tribe*. These products have a common high-level architecture but not necessarily common functionalities. Products that share the same resource platform are called a *product culture*. Such products have neither common architecture nor common functionality, but they have a common set of resource abstractions. The product hierarchy is introduced in Fig. 2.

The Symbian OS is a mobile system platform (see e.g. [5]). Its design is optimized for mobile devices, and due to its specialized origins, it is an interesting candidate for applying the layered model discussed above in the mobile systems domain. In this paper we view the Symbian OS as a software product-line according to the layered model discussed above. We aim to gain new insight to the structure of the Symbian OS and to the architectural principles

one should follow when developing Symbian OS applications. We also clarify the relationships of the different parts of the Symbian OS. We believe that a similar analysis could benefit the understanding of other software platforms as well.

The rest of this paper is structured in accordance to the layers of the above model. Each layer is discussed in a section of its own, containing both a general description of the general contents of the layer, and the features that the Symbian OS provides for that particular layer. Finally, a discussion concludes the paper.

2 Resource Platform Layer

2.1 General overview

Resource platform is the lowest layer determined in the above model. Essentially, the platform defines an application programming interface (API) for accessing the basic services and resources of the environment. The resource platform separates the software system from its external resource environment, and thus facilitates the potential changes of that environment. For the latter purpose, the resource platform layer should map the underlying resources and services of the environment to an abstract, environment-independent interface. In many conventional systems, the resource platform corresponds to the services provided by the underlying operating system. In that case the resource platform is considered as an external system that is beyond the control of the application developer, but it is still important to understand its role.

The use of the defined operations does not rely on a predefined architectural style or domain of the products. As the services are called through a simple API, the caller becomes dependent on the form and semantics of the service, but not on the internal structure of the resource platform. However, sometimes the resource platform may possess a state, in which case the order of the service calls may be restricted. For instance, semaphores and files are common resources that require particular usage patterns and conventions. In that case the resource platform should inform the caller about incorrect usage (e.g. with exceptions), allowing the caller to respond accordingly.

When designing a resource platform, one should be interested in system implied by its use of resources. Any overhead resulting from the resource platform can be multiplied when the final system has been implemented. Therefore, the resource platform is a fruitful candidate for performance and memory footprint optimization, which are two major concerns in many systems.

2.2 Symbian Resource Platform

At the heart of the Symbian resource platform is its kernel. In the simplest form, Symbian OS kernel can be understood as a context-switching machine, which does not radically differ from other commonly used operating systems that are based on a microkernel. The core properties of the system are the following. *Threads* are used as the units of execution. They are scheduled for execution in a pre-emptive fashion based on thread priorities. Unlike in some other systems, *processes* are the units of resource reservation, and act as containers of threads. Threads are allowed to share data as long as they are located in the same process, which is enforced by the associated memory management system. Any communication between two threads that belong to different processes will be handled via kernel support. This means that the designer can aim at improved performance by allocating different threads to the same process when the threads share many variables. Although the kernel provides this mechanism, there is no enforcement that would require the developer to take this option into account during design.

Table. 1. Some Symbian servers

Server	Function
Comms Server	Communications server drives the communication ports, and runs serial protocols using other communications services.
Socket server	Socket server provides a socket-based API that can be used for IrDa and dial-up TCP/IP connections.
Telephony server	Telephony server is used to control phone-like devices, enabling services like voice and data calls as well as faxing
Messaging server	Messaging server hides the internals of the messaging subsystem from the rest of the system.
File server	File server is used to hide the internals of Symbian file systems from applications.
Media server	Media server hides the details related to e.g. producing sounds in a Symbian device.
Window server	Windowing server is dedicated to handling user input and drawing on the screen.

An important issue for mobile systems is that code can be executed directly from read-only memory (ROM) without first loading it to random access memory (RAM). This in-place execution obviously contributes to the run-time memory footprint in a positive fashion. Another issue that supports the efficient use of memory is constituted with dynamically loaded libraries (DLL). They allow a memory-efficient way for managing variance regarding individual components, as it is possible to create several implementations, targeted for e.g. different protocols, and select the desired one on the fly. Moreover, many issues related to implementation mechanisms are optimized for memory footprint, including for instance the linking of DLLs.

In addition to the above properties associated with the very core of the system, an important part of the Symbian OS is the way resources are encapsulated in the system. Hardware (and also many software) resources are predominantly protected with servers (see Table 1), and therefore, client-server sessions are built in the system. Naturally, the programmer is assumed to comply with the defined practices.

Lastly, error recovery related practices form a complex set of programming conventions. The goal is to allow the designers to ensure that even in the case of an errorneous operation, the system still remains in a consistent state. This includes, for instance, a proprietary mechanism called cleanup stack, which enables e.g. freeing of memory blocks that have become garbaged and closing of client-server sessions associated e.g. with open files that otherwise would remain open forever. Using this mechanism is entirely on the responsibility

of the programmer, which requires explicit design concern to be placed on related issues. In general, of course, such a mechanism contributes to the reliability of the system.

3 Architecture Platform Layer

3.1 General overview

The architectural style of a product-line architecture is of crucial importance for a product-line. We will refer to the layer that defines the architectural style as the architecture platform. The architecture platform defines guidelines for structuring and developing systems. It provides specific component roles as implied by the style, to be played by various components of the product platforms, and points where these components can be attached. It also provides a set of conventions or design patterns that must be followed by those components.

When relying on the object-oriented paradigm, a role can be represented by a base class (interface) in the architecture platform, to be subclassed (implemented) in the product platform. Hence an object-oriented architecture platform can be implemented as a framework. Such a framework can be called *architecture framework*, to distinguish it from the more conventional *application framework*. Note that the extension points of the architecture framework are domain-independent, pertaining only to the architectural style.

The interface provided by the architecture platform may also include API-like service interfaces. However, unlike when regarding the resource platform, each caller assumes a certain role with respect to the architecture. The role may imply certain interaction patterns that must be followed, whereas the callers of the resource platform are in a uniform position with respect to the platform.

The design of the architecture platform is dominated by the quality requirements of the eventual products. The architecture platform may be developed by the same company that builds the products, or it may be a third-party component, possibly based on a standard interface.

Examples of architecture platforms could be an EJB (Enterprise JavaBeans) product supporting proxy-based distributed client-server architectural style [2] for business applications, a dispatching platform supporting the implicit invocation architecture style [5], or a graphical platform supporting the model-view-controller architectural style [2].

3.2 Symbian Architecture Platform

The architecture platform layer in the Symbian environment provides an infrastructure that can be easily adapted to different kinds of mobile devices. Applications in the Symbian environment are built by attaching user code to a Symbian-defined framework, *EIKON* [5]. The framework eases the development and the installation of graphical applications.

Like in many systems where graphical user interface plays a dominant role, the framework is based on MVC (model-view-controller) model. The model is enforced on all graphical applications. Internally, the enforcement of MVC model is handled by a component called *AppArc*, which provides the structure for all the applications. Then, *EIKON* adds some syntactic sugar for making the task of a programmer easier.

In addition to *AppArc*, control parts of *EIKON* applications are mapped to *CONE* (Control Environment) component. This component will be responsible for implementing a higher-level interface to a windowing server, which controls screen access.

For a developer, the above implies that an application will consist of at least four objects. The objects are discussed in more detail in the following.

- *Application object*, whose task is to create an instance of a subclass of document class. Technically, this object is an instance of class `CEikApplication` provided by the *EIKON* framework.
- *Document object*, which corresponds to the model part of MVC. In other words, it encapsulates the internal state of the application. In addition, the document object creates user interface. At the level of program code, this is accomplished with an instance of class `CEikDocument`.
- *User interface*, which corresponds to the controller of the MVC model. The task of this object is to handle the controls provided to the user. In addition, the object creates the view part of the application. This is implemented as an instance of class `CEikAppUi`.
- *View* is different from other parts of the application in the sense that all others are singleton objects, whereas there can be several views. The responsibility of the object is to provide different facilities for displaying the data included in the model. The class that provides such facilities is an instance of class `CEikAppView`.

Mastering all these objects can seem confusing at first, as their relation is not very explicit for the developer. Rather, one is supposed to first understand the underlying framework, and only then focus on applications. However, with some experience the developer learns their architectural roles and can fairly easily use them.

4 Product Platform Layer

4.1 General Overview

The product platform builds on the general architectural style provided by the architecture platform, and provides a skeleton for products in a particular product category. The skeleton may embody refined architectural solutions for this domain, within the boundaries of the underlying architectural style. The product platform provides a specialization interface that allows for variance in the functional requirements of the products. The design of the product platform is affected both by the (common) functional requirements and by the quality requirements of the products.

A possible form for the product platform is a framework together with supplementary components, each component being used in at least two different products. Such a framework is called an *application framework*.

A product platform is usually developed by the same company that develops the products: from the business viewpoint, a product platform allows the company to store its knowhow on building software for a particular domain, so that this knowhow can be easily reused for new products.

Examples of the product platform could be an insurance application framework based on EJB, or a network management system based on Java's Swing.

4.2 Symbian Product Platform

The product platform of the Symbian architecture[1] is constituted by the different *reference designs*. In the technical sense, all the different product categories result in libraries that are intended for mastering the capabilities of different products. In other words, reference designs

[1] In some other environments, the different product platforms vary in accordance to domains (e.g. insurance, banking, etc.). In the mobile setting, however, it is more beneficial to consider the properties of different devices rather than domains.

are predefined product configurations that have somewhat different characteristics. In the following, we will introduce the configurations of the Symbian OS discussed in [5].

Crystal is a code name for a communicator platform. The platform is very communications oriented, and includes the support for a keyboard. Products of this category have adopted a large screen, thus maximizing the benefit of the reference design. Nokia 9210 is an instance of this product platform [7].

Quarz is a platform for Symbian devices with a touch screen and features related to it. It includes a support for the recognition of handwriting as well as powerful integrated wireless communications. Therefore, it is fair to state that both Crystal and Quarz are aimed at very high-end mobile devices, where high performance and user extensibility are characteristic features. Sony Ericsson P800 is an instance of this product platform [8].

In contrast to the high-end platforms discussed above, *Pearl* technology platform is a software platform for so-called smartphones. These are intended to be voice-centric products, and to have their origins in traditional mobile phones. The goal is to give improved possibilities for reducing product complexity, to the extent that e.g. no new applications are allowed to be installed. Realizing this has led to a design where minimal facilities are offered for the manufacturer. Then, it is up to the manufacturer to design and implement a product platform that satisfies the requirements of the manufacturer.

5 Product Layer

5.1 General Overview

The product layer implements product-specific functional requirements. The product layer is written against the specialization interface provided by the product platform. This interface may hide the underlying architectural style.

It is often possible to generate the product-specific parts, or a large portion of them, automatically on the basis of a description of the desired functional properties of the product. The product layer is typically company-specific.

5.2 Product Layer in Symbian architecture

When developing applications one should use the appropriate product platform layer. Furthermore, servers introduced at the resource platform layer are also available for an application developer. However, the developer is required to follow the predefined conventions regarding the use of the servers.

The applications are typically divided into an engine and a graphical user interface part. The engine implements core application specific functionality and the GUI part handles interaction with the user. Engines are used for portability across different product platforms. One can therefore take engines as application-specific extensions of the architecture platform. The graphical user interface, in turn, is assumed to be reference design specific. Provided that the developer follows this division, applications remain largely portable across different reference designs, while still preserving device specific look and feel. Moreover, with well-established platform interfaces, it is possible to enable 3rd part development that also reuses the underlying system.

6 Discussion

We have applied a layered model for product-line systems to the Symbian architecture. Fig. 3. illustrates all the four layers discussed above. In the technical sense, the layers indicate that it is possible to separate the different design concerns in a coherent fashion.

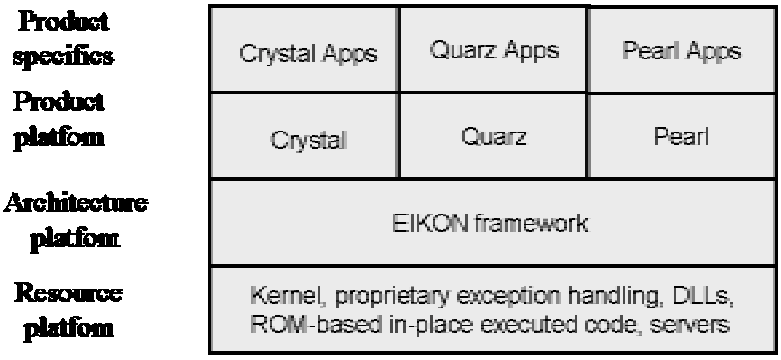

Fig. 3. Symbian software product line layers.

The different concerns that can be identified in the Symbian architecture are listed in the following. The most obvious design concern is the strict focus on low memory usage at the resource platform level, supported by both DLL mechanism and servers. At the architecture platform level, the focus is on the development of a portable yet efficient basic framework for applications. This has resulted in the use of MVC-like architecture for applications, and therefore, the model parts of applications can be reused in different products. At the level of product platform, the variance is based on the different physical characteristics of different systems, including e.g. varying screen sizes and input mechanisms. At the level of individual products and applications, the developers are responsible for putting the right underlying mechanisms to use.

Of particular interest is that the layering scheme discussed in this paper is not a direct derivative of the conventionally advocated layering approach based on levels of abstraction. Rather, the layers we have discussed in this paper have more to do with the variance of the different products. This means that when deriving such a product line for mobile systems, one needs to place the emphasis on the different products that are supposed to benefit from the architecture, not only technical issues.

The amount of explicit attention paid to the discussed issues during the design of the Symbian OS is unclear to us. However, based on the discussion in [6], many decisions contributing to the layered arhitecture have been conscious, although not necessarily aiming at the structure.

References

1. Bosch J., *Design & Use of Software Architectures: Adopting and evolving a product-line approach*. Addison.Wesley, 2000.
2. Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Son Ltd, 1996.
3. Clements P., Northrop L., *Software Product Lines: Practices and Patterns*. Addison.Wesley, 2002.
4. Myllymäki, T., Koskimies, K. and Mikkonen, T., Structuring product-lines: A layered architectural style. *Accepted to OOIS'02, to appear*.
5. Shaw M., Garlan D., *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
6. Tasker M., Allen J., Dixon J., Shackman M:, Richardson T., Forrest J., *Professional Symbian Programming: Mobile Solutions on the EPOC Platform*. Wrox Press Inc, 2000
7. Nokia 9210 documentation. *Available at http://www.nokia.fi.*
8. SonyEricsson P800 documentation. *Available at http://www.sonyericsson.com.*