Architectural Evolution

Nokia Mobile Phone Case

Juha Kuusela
Nokia Research Center, Helsinki, Finland
juha.kuusela@research.nokia.com

Key words: Architectural evolution, product family, and organization.

Abstract: Similar software products can be developed as a product family. Common architecture, addressing all common requirements of products in the family, provides the basis for wide scale reuse within the family. When independent products continue their evolution, they face new requirements that may prove to have wider scope and need addressing at the family level. However, changes on the family level may be very costly for the product projects. Our experience shows that architectural evolution is possible and practical if each change has been carefully planned, taking into account its organizational aspects. Then the change has to be carried out so that the product line does not stop. Large architectural changes are high-risk operations; even when they succeed, they tend to take much longer than expected.

1. INTRODUCTION

The software architecture group at Nokia Research Center was established in 1994. Our group has now 17 members and operates both in Helsinki and in Boston. We lead two international research projects ARES and FAMOOS within the Esprit program but our main task is to support Nokia business units in their product development. With the business units we analyze, assess and model their product architectures and give suggestions on how to improve them. We also participate in developing architecture for new product concepts.

In the cooperation with Nokia Mobile Phones (NMP), our role has been to facilitate the process, introduce state of the art in architectural design and description to software architects from NMP, review and comment their

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35563-4_35

P. Donohoe (ed.), *Software Architecture*

designs. The final architectural choices and their implementation have always been the responsibility of NMP architects. During these 4 years, we have had an opportunity to observe how software evolves in response to changing requirements and to learn how this evolution affects a software development organization and its development process. This experience report gives an overview of this process, presents examples of architectural evolution, and offers a classification of different architectural changes and an observation on how difficult they are to implement.

Nokia Mobile Phones produces a range of similar mobile phones. It has an opportunity to control the properties and quality, and to reduce the development, maintenance, support, and marketing costs of each product by sharing some of the effort and parts between these phones. In order to manage such sharing, the phones are organized into a product family. There are many reasons for variation in the mobile phone family. Different market segments have different characteristics and the products must offer a choice of functional features and capabilities to satisfy a wide spectrum of customer requirements. National standards often impose constraints on product functionality. Cultural differences and fashion add variation to the user interface design. Advances in technology require frequent migration of products to new platforms and environments.

Software architecture provides the basis for reuse within the product family but it also ties the products together and limits their evolution potential. Architecture can only be designed to accommodate anticipated variation. Some of the reasons for variation in the mobile phone family are rather stable (like different languages) but most are volatile and can only be anticipated few years ahead. Once the architecture can no longer support the product family, it has to be changed.

Architectural changes can be very costly. Much work is needed to update everything and changes in the basic premises may force redesign of large parts of the system. There are many sources of architectural changes; some changes can be avoided by careful planning but others are unavoidable. If the products based on the architecture are successful, new products with new properties will be added to the family. Architecture has to be periodically updated to support the new needs. This paper summarizes the architectural evolution of Nokia mobile phone product family.

2. NOKIA MOBILE PHONE FAMILY

Traditionally a cellular phone consists of a transmitter and a receiver for communication with the network, a user interface consisting of a keyboard and a display, a battery, a microphone, and a speaker. In addition, the phone

has a processor and memory for the software needed for controlling the hardware. The phone may also have facilities for some auxiliary services, such as data communication.

A cellular phone communicates through the cellular network. Nokia has developed phones for various network standards, e.g., for analogue standards NMT, AMPS and TACS, and for digital standards such as the Japanese JDC and the European GSM. TDMA and CDMA are adopted in North America. For each cellular standard, Nokia provides several phones for different market segments. These phones vary in style, functionality and price. The variation is implemented both in the hardware and in the software. The development organization is large and globally distributed.

The complexity of the product family, the structure of the development organization, and the need to introduce new features as they become available in the networks, makes mobile phone software development a challenging task. Hardware development is the basis for competition but in order to benefit from this potential, new phones have to be on the market before the competitors' models using the same hardware. Software development time costs money.

So far, the evolution of the mobile phone has had only a few basic drivers. *Miniaturization* has shrunk the size from portable (like a suitcase) to devices weighting less that 100 grams. At the same time *operation-time* with a standard battery has grown from hours to weeks and *price* has dropped from the status-symbol level to that of an affordable personal phones for each family member. Through this evolution, the product concept has been rather stable: mobile phones are used for voice communication. Now we also see evolution of the product concept. The phone has an increasing role as a portable terminal to an information system and as a communication device between information systems. Intelligent add-ons see the phone as a center of a distributed system, car electronics as a general-purpose communication device, and Internet based systems as a portable browser. This change in the product concept has a large impact on the product structure.

3. INITIAL ARCHITECTURE AND DEVELOPMENT PROCESS

Initially, the software architecture of the phone addressed only the basic requirements variation in the hardware, the communication standards, and the user interface. These domain characteristics formed the basis for the initial module architecture and this architecture had only three subsystems:

1. A cellular subsystem for managing the connection to the network.
2. An application subsystem that includes the user interface software.

3. A device subsystem for interfacing with the hardware.

The separation of the cellular subsystem is critical since it allows easy development of different phones for each network and similar phones for different networks. Separation of the device subsystem is also crucial to be able to benefit from constant hardware evolution.

The development process for this architecture was very product centered. The development organization would base each new project on some earlier version of the subsystems and make the necessary modifications. This allowed each development organization to be rather independent supporting the rapid growth of a distributed organization.

4. TENSION IN THE INITIAL ARCHITECTURE

The initial architecture and the development process were very successful. However, the architecture had many inherent problems. The very separation of application software from cellular subsystem creates a problem. Different network standards have different capabilities and thus application software is in reality coupled with the cellular subsystem. The coupling is visible in the specifications since some user interface applications involve complex protocols and their specification is included in the protocol standards.

The subsystems are too large. Divide and conquer does not really work well if you only divide by three. Naturally subsystems have internal structure but their interfaces are handled on a subsystem level. Consequently, subsystem interfaces grow very wide. Wide interfaces and dependability between the cellular subsystem and the application subsystem leads to high coupling between them. The device subsystem does not suffer from the same problem because it is just a composition of rather independent hardware drivers, each having its own interface.

The subsystems are not equal. Hardware drivers and cellular software have a clear role. The application subsystem is "everything else". It becomes the controller of the phone, knowing its global state. This creates state coupling and even content coupling between the subsystems. Finally, phones are not as homogenous as assumed by the architecture. Some have special functionality (e.g., data communication) to be accounted for.

5. EVOLUTION

Initially the variance in the subsystems was mainly functional. Some phones had special features and accordingly there was a special part in the

software handling it. Then the coupling between application subsystems and others started to play its role and increasingly variance in one place in the software was just a reflection of variance in another part.

Required configurations were implemented by using a configuration management system together with more fine grained mechanisms like source-code pre-processing using macros and compiler flags, or programming language based mechanisms like indirection and late binding of functions, variables and types.

The elements of variability supported by these mechanisms (text lines, functions, variables, and files) are not the elements of variability required by products (features, platform differences, interface styles). In particular, source-code preprocessing using compiler flags is problematic. It is the most versatile variance mechanism, allowing the possibility of making every source line a special case, but it does not build any abstractions. From the source, it is practically impossible to determine what each flag means, or what combinations of flags are permissible.

As the variance kept growing, it became hard to control the mapping between desired product variance and its implementation. This happened in the golden years of artificial intelligence and the "natural" solution was to automate this mapping by developing an expert system setting the compiler flags based on a list of features that the phone was supposed to have.

At the same time, the development organization was growing and subcultures started to emerge. Since each site was mainly responsible of development for a particular market area sites did not have to deal with variance in network standards. They started to maintain their own versions of application subsystems to get rid of the variation caused by multiple network standards. This confined the reuse benefits into small groups of products but it did also cut the cost of reuse and increased independence of each site.

The initial architecture proved to be very stable. A number of small subsystems were added as the mobile phones got new functions but the initial architecture maintained its central role over several phone generations.

As the phones kept evolving, new functionality was added. Now that the application subsystem was diversified, it became apparent that the cost of porting new functionality across the product family is substantial. Maintenance problems with the application subsystems also made it clear that the application subsystem had to be redesigned to be more flexible.

The redesign was carried out according to an object oriented user interface design style separating control, presentation and functionality (à la MVC). The redesigned application subsystem replaced the old application subsystems and an attempt was made to keep its interface as backward compatible as practical. This attempt succeeded. The development was

carried out parallel to the product development based on the old application subsystem.

When the new application subsystem was taken into use, the organization had to be restructured accordingly. We identified three different development categories:

1. The infrastructure development group improves the application framework and ports it on different hardware platforms.
2. The component group develops reusable presentation and application components.
3. The product development projects compose their application subsystems using existing components and develop new components when necessary.

The whole process is driven by the product development projects. They place requirements on the infrastructure and request new components.

Up to this point, we had been able to accommodate each change either by adding new modules to the architecture or by reworking existing elements. Recently we had to face a bigger challenge.

Markets continued to develop and new product ideas were put into practice. This led to erosion of the basic premises underlying the initial software architecture. We could no longer assume that there would be only one cellular system in each phone since phones should be able to operate in different networks. Auxiliary equipment continued to become more intelligent. In addition to the user interface, phones could be controlled by infrared and serial connections. In one setup, the phone had to act as a central controller for a large distributed system and in another it was completely subordinate. The success of the Java programming language also pushed downloadable software to phones. Clearly, we needed a new architecture.

The basic idea behind this new architecture is to separate the service identity from the identity of its provider and make service usage and provision location independent (see *Figure 1*). With dynamic configuration management, we can have several providers for the same service and these providers can be plugged in or taken out without restarting the system. Architecture supports both local and remote message passing and object management, task scheduling and event control. This architecture is also much better described. It defines software components, message interfaces between components, essential use cases, component grouping and deployment structure. The initial architecture had only interface and runtime architecture descriptions.

The biggest challenge in this new architecture was not how to design it but how to adapt to it without stopping the product line. We approached the problem by moving into the new architecture gradually. A roadmap of new

architecture versions was outlined. Each version has more capabilities than earlier ones. Architecture versions are developed concurrently with product development projects and each project is based on a version that satisfies its needs. Currently our new phones are based on the second version; a third version is being implemented, and fourth is under design.

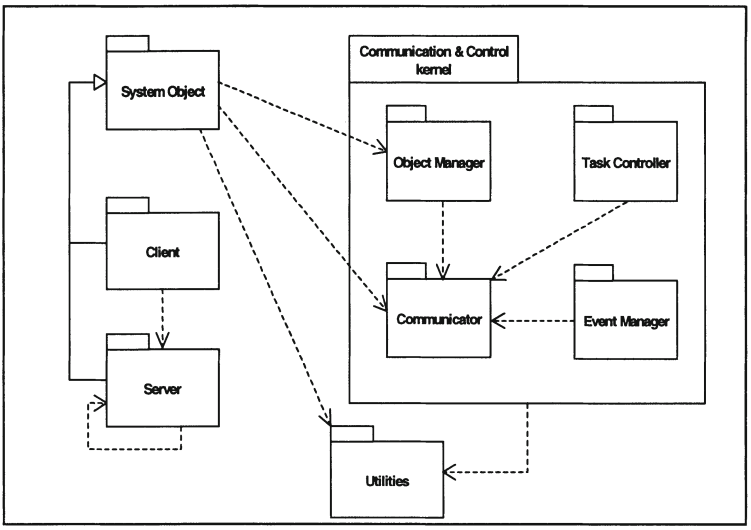

Figure 1. Basic module classes for the new architecture

This architecture evolution roadmap that was planned to help us to control the move from the old architecture to the new one is going to be permanent. It gives a view of the future and helps the product development projects to assess what will be possible and when. It is also a basis for reasoning about how to develop further the product family.

6. LESSONS LEARNED

It is often assumed that the development and management of an architecture addressing all the common requirements of a product family and providing the basis for wide scale reuse would always be economical. This is not quite true. When independent products continue their evolution, they face new requirements. These requirements can be tackled only in the product development project that control the resources and have the responsibility. Later, some of the new requirements may prove to have wider scope and they can be tackled on the family level. However, changes on the

family level may be very costly for the product projects. Commonality management also requires communication and cooperation. Such a cooperation between different organization over wide distances is complex and costly. Reuse and modifiability must be balanced according to the product development organization and market needs.

Our experience shows that architectural evolution is possible and practical. We made three different types of changes.

1. Adding components, which turned out to be rather easy as long as they required no special services.
2. Redesigning components, which was difficult whether you changed the interface or not.
3. Redesigning the architecture with new communication mechanisms, a new execution architecture, and new component roles, which was very difficult and costly.

Note that all the changes were incremental; nothing was ever built from scratch.

This experience shows that architectural change has to be carefully planned, taking into account its organizational aspects. Then it has to be carried out so that product line does not stop. Large architectural changes are high-risk operations. Even when they succeed, they tend to take much longer time than expected. New products cannot wait for the new architecture.

This history also demonstrates that variation management and reuse are tightly connected. If your variation management runs into trouble, you may ease the situation by decreasing reuse. The gray area between a perfectly organized product line and completely independent development projects is wide.

It takes time to react to the changes in the domain of requirements and a practical product line is never optimal. Make a roadmap showing what you intend to implement and when. Base the changes on the needs of product projects and product concept developers. It is hard to give reliable economical justification for each change but it is easier to compare different change requests.

ACKNOWLEDGEMENTS

I wish to acknowledge the help provided by my colleagues Alessandro Maccari, Anssi Karhinen and Alexander Ran and anonymous reviewers. Their comments improved this report substantially.