Experiences with Software Product Family Evolution

Claudio Riva and Christian Del Rosso
Software Architecture Group
Nokia Research Center
P.O. Box 407, FIN-00045
Helsinki, Finland
{claudio.riva, christian.delrosso}@nokia.com

Abstract

The evolution of product family typically oscillates between growing and consolidating phases. The migration path starts from a copy/paste approach that offers the fastest time-to-market and then moves towards a mature software platform that offers a higher throughput of products. We have identified several issues that harm the evolution of the family: new requirements that can break the architectural integrity of the family, increasing level of bureaucracy in the organization and a slower process of change. In this article we present two approaches for coping with the family evolution: architecture assessment and architecture reconstruction. We also present Nokia case studies where the methods have been successfully applied.

1. Introduction

The concept of software product family originates from the hardware industry where hardware product lines[1] enable the production of numerous variants of products and a significative reduction of operational costs by sharing most of the assets. In the recent years the software has become a dominant part in an increasing number of embedded products and it is often affecting the quality and the delivery time of the products. We can estimate that most of the delays in the release of embedded products is due to software rather than hardware faults. To cope with the multitude of software variants required by an industrial product line, the software assets have been organized in software product families and, thus, the paradigm of product line has been transferred to the software embedded in the products. This paradigm shift has happened for most of industries producing embedded products (like cars, consumer electronics and mobile phones) where they need to deliver a customized software system for the various products.

A software product family is a collection of products that share common requirements, features, architectural concepts, and code, typically in the form of software components. In this article we investigate the problems that affect the evolution of a product family and we introduce two techniques that we commonly use for coping with those problems: *architecture assessment* and *architecture reconstruction*.

2. Product Family Evolution

Software product families are rarely created right away but they emerge when the domain is mature enough to sustain the long-term investments. The typical pattern is to start with a small set of products (often just one). If the business starts to generate profits or looks profitable in the future, new products are introduced in the market. New products are typically copy pasted versions of the existing ones with some additional new features. Most of the differences are achieved at the software level, while the hardware platform remains quite unchanged. On the wave of success, the software embedded in the products becomes a global asset that becomes in use in several sites worldwide at the same time. Many sites embed the same software in their local products and often make their own local modifications (e.g. customization, updates, patches). As soon as the business becomes more mature, new investments are needed for consolidating the software assets. At this point, the various set of products are migrated towards a product family in order to keep all the software variants under control. The migration process affects the software parts and the organization as well. The organization needs to adjust its operating procedures to support the global management of the products lifecycle (from requirements engineering to testing). The software variants have to turn into a flexible platform where the products of the family can be derived from in a more

[1] For the discussion in this article software product line and software product family can be considered synonymous.

Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE'03)
0-7695-1903-2/02 $17.00 © 2002 IEEE

flexible way. We can identify five different patterns or approaches that appear at different stages of the evolution:

copy/paste: the software variants are created by copying and modifying the existing products. It is the fastest approach for creating a new product and it is typically used when the organization is entering or creating a new business. All the resources are focused in the implementation of new features without a strict control redundancy that the variants create. This approach minimizes the risks of development but it maximize the entropy. This approach gives the highest flexibility for creating new products and entering in a new market. Verhoef et al. describes this process as *software mitosis* [8].

configuration: The variability is embedded in the software with a set of configuration parameters. The parameters allow to enable/disable parts of the code, to select particular algorithms and to configure the modules. Although the method is simple and allows to create numerous variants from a small set of bases, it has several drawbacks concerning the maintainability and evolvability of the code.

component-based: the variable functionality of the software is factored into separate software components and assigned to different development teams with a clear separation of concerns. The variants are achieved by plugging different components into a common software framework. The granularity of the components can range from single classes to entire subsystems. The correct granularity is often a trade-off between the flexibility/maintainability: few large components reuse more software but are harder to compose and maintain, small components might embed too little functionality. The correct size is often reached after some time. The component-based approach has an impact on most of the software engineering activities of the organization, especially for the integration phase of the components in the products. The component-based approach can be considered a key milestone towards a flexible product family.

platform: the concept of software platform emerges when the organization starts to consolidate its experience in a mature domain. The goal is to maximize the reuse of the software components among the products and the throughput of the family. The platform provides a cohesive set of services, libraries, software components and product frameworks that are used for building the products. The basic services (e.g. telecom protocols, hardware drivers, graphical libraries, common applications) become globally available in a precise and controlled way. But that's not all. The platform becomes a well-defined entity in the organization with its release plan, roadmap for the new features, coding conventions, idioms, testing procedures, architectural documents, training material. The development teams are organized in a matrix structure. On one dimension there are the *component factories* and the *platform management* team responsible for the development of the software components and for the maintenance of the platform. On the other dimension there is the *product development* organization responsible for the development of the products. The crucial phase of the product development is the integration where the different components have to be integrated in the coherent way.

optimized platform: the optimized platform tries to overcome with the integration problems of the platform approach. Most of the resources are spent during the integration of the platform components when architectural mismatches or bugs have to be carefully analyzed and solved by the component owners. The optimized platform solves this problem by enabling the feature-based derivation of the products. The platform offers a rich set of configurable features. The product integrator selects and configures the features to be included in the product and the real integration is automatically achieved by the platform. This has been envisioned in our previous work [13].

In real product families the five approaches can coexist at different extent. In a highly dynamic domain, the product family is more directed in the direction of copy/paste approach that offers the fastest time-to-marker. In a stable domain, the platform approach is a better choice because it maximizes the consolidation of the assets. Verhoef et al. describes the same concept using the *grow and prune model* [8]. The product family is typically oscillating between grow and prune phases. In the grow phase, the product family is free of exploiting new opportunities without much architectural governance using the copy/paste approach (this leads to the software mitosis phenomena causing a large increase of clones and variants). In the prune phase, the weak branches of the family are removed and the successful products are re-organized in order to be consolidated in the family (re-balancing the robustness and the governance that was lost during the mitosis phase).

3. Product Family Architecture

We present a typical example of a Nokia's product family architecture. The main goal of the product family architecture is to describe the commonality and variability of the family in order to make explicit the variation points of

the products. We use a conceptual hierarchial framework for describing the elements of the product family architecture. We can identify at least four layers of genericity: the *reference family architecture* layer, the *family architecture* layer, the *lead product architecture* layer and the *copy product* layer.

The reference family architecture describes the global architectural style that is valid for all the products of the family: architectural significant requirements, architectural rules, patterns, component types, communication infrastructure, runtime issues. The architects can derive the software architecture for the product families from the reference family architecture.

The mobile phones are grouped into product families according to the UI styles, features, telecom standards and hardware generations. This represents a first level of variations where the products are grouped in macro-families (e.g. GSM, TDMA or UMTS). For each product family, the family architecture describes the services and features that are available in the platform. They may include the protocol stacks, the OS, the UI kernel, basic applications and hardware drivers.

Each family contains a reference product implementation that we indicate as lead product architecture. This product is considered to be the most typical one of the family. It is derived from the family by copying the common elements from the family architecture and by instantiating the abstract elements. The purpose of the lead product architecture is to provide a reference architecture for the other products and to clearly document the variations points available within the family.

At the bottom of the hierarchy, there is the copy product. This is typically copied from the lead product and adapted to the specific product requirements. This represent the final product architecture and it is the starting point for the development project (mainly focused on feature configuration, integration and testing).

4. Product family issues

In the Section 2 we have presented five different approached for organizing a product family and in the Section 3 we have presented a typical product family architecture. In this section, we discuss several problems that typically concern the evolution of a product family.

4.1 Increasing Bureaucracy

The migration towards a product family is a process that introduces bureaucracy in the organization. The software process becomes more complex due to the introduction of new procedures that have to be followed when creating a new product or modifying the platform. Differently from the uncontrolled growing phase, changes have to be well documented and motivated. It also emerges a new hierarchy of managers, architects, feature owners, component designers that are responsible for preserving the integrity of the product family architecture and for approving the changes. Enforcing the architectural governance requires a certain level of bureaucracy but this is also a threat for the flexibility of the family. This tendency towards stiffness is often opposed by practices that increase the flow of communication among different teams, for example by introducing architects that are responsible for heterogenous technology areas.

4.2 Slow process of change

There are cases when the change requests for new features have to go through a long approval process. If we consider the four-layer architecture of the Section 3, a typical scenario is the following. A new feature is detect by the product development team at the lowest level. If it is a local feature and it does not have an impact on the family, it is just implemented in the local product. If it has a possible impact on the family, the feature has to be passed over and over to higher levels where its impact is carefully assessed. In the worst case a change request may reach the reference architecture level. This happens when the new feature requires a critical change at the core of the family (for instance, adding a streaming video functionality might require changed in the operating system). At some point the change request may be reject or delayed to avoid the negative impact that its implementation would have on the architecture. A slow process of change is an inevitable drawback for avoiding features that could break the architectural integrity of the family.

4.3 Over-designed platform

The design of a new software platform is a long-term activity where considerable resources are spent for designing a generic-enough platform to support the long-term evolution of the product family. There is often the risk of designing a platform that is too generic for what is really needed by the products. There is a sort of auto-inducted tendency of searching for the best software design that can handle all the possible situations. This often leads to the creation of far too complex software frameworks that are very difficult to instantiate. This tendency should be limited and the design activity should investigate the good-enough architectures rather than the best solutions.

4.4 Spaghetti dependency

A main goal of a product family is to share software among several products. Since the owners of the software

components (i.e. the component factories) and the users of these components (i.e. the product development teams) are different, for each product there is an inevitable network of dependencies. Common problems are: the interfaces of the components change without notice, long queues for the change request of widely used components, the clients of the components are unknown (the dependencies are often only visible in the code). Moreover, software dependencies can be easily mapped to human interactions among different development teams and in a multi-site geographically distributed environment managing these interactions is a challenge. Minimizing and controlling the software dependencies of the family is a key activity for the organization.

4.5 Feature reallocation

In the typical scenario the features of the product family architecture are instantiated in the specific product architecture. However, during the consolidation phase the features in the products can be re-allocated to the platform. In this case, it is necessary to move the implementation of the feature out of the product and integrated it with the platform. This often happens when a feature that has been exploited in one product has been successful and, thus, other products want to use it. In this process, we need to ensure that the feature can be supported in the entire family.

4.6 Cross-family reuse

There are cases when it is necessary to share software components among different product families for reducing development costs (for example, when migrating one product family to the latest hardware that is already in used by another family). The first problem is that there can be architectural mismatches among the families (e.g. different operating systems) and these differences have to be assessed. The second problem concern the ownership of the common software. In many cases, it is possible that the product family has little influence on the software development somewhere else. This situation often leads to a long integration.

4.7 Introduction of new requirements

In a dynamic market it is critical to handle the forthcoming requirement in time. Even though the problem of incorporating new requirements is not specific to product family architecture, the process has to accomplish an even more difficult task. The variability of the products must be considered when evolving the architecture and it must be carefully verified if a requirement for a product can lead to break the product family architecture. In the analysis of the forthcoming requirements must be ascertained how easy is to add them to the current architecture and estimates the work needed for the implementation.

5. Software Architecture Assessments for Product Families

Architectural assessment is an essential part of the system architecting process that is targeted to evaluate the degree of fulfillment of quality, or non-functional, requirements. Recent research has focused on the application of architectural assessment to software systems [1], [2], [3] as well as to software product families [9], [12].

Currently, the literature about the issue of evolution is scarce (with exceptions such as [5]), and there is no established best practice that guides into this particular discipline.

Several architectural assessment practices and methods exist. Examples are ATAM [11], SAAM [10] and experience-based assessments [4]. None of the existing methods is specifically tuned for product family architecture. Also, case study reports from industrial settings are few.

Product families include products that share common requirements, features architectural artifacts, and components or simply code. The business reasons behind architecture assessment can range among the following:

- Evaluate and improve the architecture of certain software system, with special focus on qualitative attributes.
- Evaluate the conformance of a software system to standards.
- Check whether certain qualitative requirements are satisfied by the product family architecture.
- Identify the skills needed for implementing the system.
- Validate the partitioning for implementing the system within a certain organization.
- Identify the risks related to a particular architecture.

Many of these are believed to be important side benefits of assessments, which cannot be properly classified as goals. However, these beliefs have not been experimentally proved. The input of the architecture assessment is (obviously) the available documentation and knowledge about the architecture. Its primary outcome is the assessment report. Optimally, the defects and shortcomings identified during the assessments and captured in the report lead to an improvement of the architecture and of its documentation. In the specific case of product families, architecture assessment is usually done for different business reasons. Software product families are designed to support several products bearing different features. To ensure fast product derivation, the software that is common to a certain product family is ported in new products. This emphasizes the

role of assessments, as in the case study that we describe below. We describe a case study that we performed in order to evaluate the capability of a large software product family to evolve and support certain new (major) requirements.

5.1 Case study

The scope of the assessment included a fairly large subset of Nokia's mobile terminal product family software. Currently available mobile terminals have reached a considerable level of sophistication. However, with the evolution of the mobile networks that is foreseen to take place in the next few years, a number of major new requirements will have to be incorporated. Such requirements include, but are not limited to, the addition of several new applications (especially in the multimedia domain) and the support for new hardware and communication protocols. Moreover, the speed of launch of new features is (and is expected to continue) increasing, and the product life cycle symmetrically decreasing in duration. This poses an emphasis on flexibility and modifiability of the software product family architecture. The main objective of the assessment was to estimate the capability of the existing architecture to incorporate such new requirements, and to highlight possible problems in this process. Our first step was to define the scope of the architectural assessment that was to take place. The software system we assessed is very large, and it is being developed in different sites by teams that do not regularly communicate. Architects and developers work in a distributed fashion on product development. We identified the number of stakeholders for the architecture to be in the order of several dozen. The method we had applied in our previous assessments [12] consisted of a scenario-based walkthrough that took place during a brainstorming session (meeting). This method has proved to be effective when the stakeholder team (that performs the walkthrough) is of reasonable size, so that communication during the meeting can happen smoothly. However, we had to face many stakeholders who had seldom met before the assessment, and were located in separate, often very far sites. Additionally, most stakeholders were believed to be expert of a relatively small domain area (e.g. system performance, usability, impact of new requirements, low-level software). For these reasons, we chose to perform personal interviews with every stakeholder lasting about two hours and focusing on the issues to which each of them could contribute (as estimated to the best of our and the chief architect's knowledge). The interviews were semi-structured, and during the course of the dialogue the interviewer guided the interviewees based also on information gathered previously. The interviews were intended to be as open as possible, since we needed to elicit opinions about highly technical issues that depend on both the architecture and implementation of certain feature sets in the product family. As a first step, we had to define the order in which the stakeholders would have been interviewed. Therefore, we divided the stakeholders in three broad categories, to interview in this order:

1. Those responsible for collecting forthcoming requirements and evaluating their potential impact on the architecture.

2. Experts on a specific part of the software product family or on a key quality attribute.

3. Those responsible for the development and maintenance of the software product family architecture at a high level.

We strove (as much as logistics allowed) to interview people from these three categories in a sequential order (i.e. people from category a before people from category b, and people from category b before people from category c). We associated the completion of each category with a milestone of the assessment project. The first phase included the elicitation of key new scenarios. From the interviews, we elicited the current and future requirements about the architecture. The result of this phase is a list of scenarios for evolution. These usually consist of one or more questions such as "how is the architecture going to evolve to support requirement X?" The understanding of the requirements allowed us to establish and plan the list of the stakeholders to be interviewed during the following phase. The final result of the assessment was a prioritized list of the technical and organizational issues that emerged across the whole assessment. The document containing the requirements and their estimated impact also was handed as a deliverable.

5.2 Results

Our case study was essentially an experience-based assessment. The result consists of a document that lists of potential problems in the current product family architecture. Among other things, the document contained a set of views (in the IEEE 1471 sense) of the current architecture. Since the assessment focused on evolution, we had to dedicate a fairly large fraction of the total effort to eliciting new key scenarios and periodically checking the assumptions and future requirements. We did this all the way through the assessment, based on both the existing documentation about the architecture and on the material extracted from the interviews that we had already performed. We structured the report according to the problem domain. Under each subdomain, we described the status of the architecture of the corresponding implementation, all the issues that the interviewees reported, as well as the (ongoing and advocated) improvement activities in the different development units.

We tried to perform an analysis of the issues that were presented, rather than simply reporting what we heard from the stakeholders. The report concludes with a series of hints on potential risks and weaknesses. In addition, we extracted a table (tightly integrated with the report) containing about 65 potential action points. Every action point consisted of the following elements: business problem, problem, solution, components affected and "real action". The business problem highlighted the main business reason that justifies spending resources on solving the problem, (e.g. better user experience). Every problem could have several solutions, which we categorized by means of the potential impact on the architecture (including but not limited to the number of components affected). The "real action" was aimed to management in order to address the problem, or further study unclear issues. This ensured that adequate follow-up activities would be started and associated to a responsible person and a deadline. The activities springing from such action points would become the practical impact of the assessment on the organization.

5.3 Lessons learned

In this section we will present a number of interesting lessons that we have learned during this case study. We will put them in form of advice, trusting to provide valuable material for other applications of similar methodologies. The fact that our case study was fairly well focused helped us in understanding the domain relatively quickly, and in building on the information that we had gathered during previous interviews to gradually refine the material. The structure of the interview forced stakeholders to concentrate on very specific issues, and instances where different stakeholders repeatedly quoted a certain issue as a problem increased the validity of their claims. In order to guarantee focus, the scope of the assessment must be clearly described and explained before each interview. We gave every stakeholder the opportunity to comment on and refine it if they wished. Before the interviews, we also gave a brief overview of the assessment methodology, and specified what we expected the outcome to be, thus giving the stakeholders a somewhat clear idea of the entire process and of the practical impact and value of their contribution. The main purpose of the assessment task was to identify and highlight possible weaknesses of the product family architecture. However, almost all stakeholders rated its role as a communication vehicle as very important. This is consistent with previous research work on the subject, but was particularly evident in our case study, where the size of the architecture under assessment and the geographical and organizational distribution of the development and maintenance departments naturally hampered communication. In more than one instance we came into a stakeholder who quoted a problem that had already been fixed elsewhere, evidently without proper communication. The assessment partially helped overcome this problem. Moreover, the report we produced still serves as a sort of "white paper" on the status of the product family architecture and the main problems it suffers from. In several instances we have noticed different opinions between architects on where certain problems originated and on the methods that could be used to solve them. In a few cases we were proposed improvement activities without a clear statement of what problems they were meant to solve. All the claims must necessarily be based on the technical evidence supported by facts and, possibly, measurements. If no proof supports certain claims, then follow-up activities must be undertaken in order to define the problems in a more objective way. For instance, it is not enough to hear that some product has low usability: you have to perform simulations with real users to evaluate this specific quality attribute. During the course of our interviews, we noticed that some stakeholders tended to minimize the relevance of the problems, thus assuming an "everything can be solved" attitude. Others, however, were more pessimistic, implying that "the whole company will suffer from this if we don't do something soon". As always, the truth lies somewhere in the middle. The assessment coordinators must try to keep balance and report facts as they are, with as little emotional bias as possible. For this reason, it is important that the assessment team be part of an external organization. When the assessment coordinators are part of the organization that is responsible for developing the architecture, their opinion is naturally subject to more biasing and influence. The reasons can be various, from trusting your close colleagues words more than others to having a vested interest in promoting a positive evaluation of the architecture. All these situations must be avoid during assessments. For the reasons we list above, we recommend (and will keep on) promoting architectural assessments as a periodical task during the architecture development life cycle. Our as well as other researchers' previous case studies provide strong evidence that the benefits of assessments far surpass their cost. We advocate the continuation of research in the field, as well as the publication of more industrial experience reports. These should also aim for a quantitative and qualitative comparison of interview-based versus brainstorming-based scenario walkthrough methods.

6. Architecture Reconstruction and Conformance Checking

The evolution of a family is mainly driven by two forces: the consolidation of the assets in the platform and the creation of new products. The platform slowly evolves by incorporating the new architectural requirements, while new products are added by introducing new features. This ap-

proach is supported by a combination of forward and reverse engineering. Forward engineering activities are necessary to develop the new features during the grow phase. Reverse engineering [6] activities are mainly concerned with the consolidation of the assets:

- recovering the updated product architecture (as opposed to the intended architecture that was in the minds of the architects)
- monitoring the organization of the components in the platform
- coping with the architectural dependencies within the platform and among products.
- enforcing the conformance to the architectural rules

The main goal of our architecture reconstruction method is to recover architectural models that the architects can use to comprehend the actual implementation of the products. The focus of our reconstruction is mainly on the architectural significant aspects of the products (e.g. the logical dependencies among the software components).

Architecture reconstruction [14] deals with the task of recovering the past design decisions that have resulted in the present implementation of the system. The design decisions are about the concepts that represent the building blocks of the system and the structure that describes how the different software entities are connected. Similar to archeology, which aims at studying man's past through scientific analysis of the material remains of his cultures, architecture reconstruction is a reverse engineering activity that infers the architectural rationale from the available artifacts (e.g. code, design documents, interviews with the system experts).

The outcome of the reconstruction is typically presented using multiple views [7] that show different aspects of the architecture:

Component view: describing the major components, their interfaces and their logical relationships

Task/Process view: describing the task allocation of the architectural entities and showing the inter task communications

Development view: describing the organization of the source code files and their relationships (for example, include dependencies)

Deployment view: describing the physical location of components in the processing units.

Feature view: describing the run-time implementation of a feature at a high level of abstraction.

Organizational view: describing the organization of the development activities (projects, programs, sites).

A more detailed description of the reconstruction method can be found in [14]. In this section we discuss the product family aspects of the reconstruction and present two case studies.

The reconstructions activity starts with the identification of the architectural concepts of the product family. Every software system is built according to a particular architectural style. The style defines the types of building blocks that can be used to compose the system (e.g. components, classes, applications) and the communication infrastructure that enables the components to interact at runtime (e.g. software busses, remote procedure calls, function calls). Those concepts represent the way developers think of a system, and they must become the first class entities, the terminology of the reconstruction. The reference architecture describes the global architectural style for the family and it is typically the source of information for this phase. In the case the reference architecture is not available, it has to be recovered from the existing products. This phase is conducted with the experts of the system through a series of scenario-based evaluations of the system. During this phase, the architecturally significant requirements (ASR) of the family are detected. It is important to re-document them because every architecture has been designed to support some specific requirements. Understanding the ASRs allow to understand the motivations of certain design choices.

The second phase concerns with exploration of the software and starts by extracting a raw model of the system. A raw model is a collection of basic facts known about the system at a low level of abstraction. The facts can be extracted with a variety of methods: lexical or parser-based tools for analyzing the source code, profiling tools for the dynamic information (e.g. message passing, process spawning, inter-process communication), manual analysis of design documents, and interviews with the developers. Not all the facts are directly available and some must be inferred.

The raw model is typically a large and unstructured data set (containing tens of thousands of entities and relations). We can enrich the model by classifying and structuring the model elements in hierarchies and by removing architecturally irrelevant information. The classification is conducted by interviewing the architects. For each logical component, we have identify its counterpart in the reconstructed model and we add the mapping to the model. In the final model, the logical dependencies among the software components should become visible. The model is typically presented using the multiple views previously discussed and using different presentation formats (UML diagrams, hyperlinked documents, graphs).

The final step is to check the conformance of product architectures against the architectural rules. The reference ar-

chitecture and the family architecture contain several rules that have to be valid for all the products of the family. Once we have extracted the updated architectural model we can check for this conformance. This check can be automated and executed periodically to identify the products that are violating the rules.

6.1 Case study 1

We have tailored the reconstruction method for the analysis of a Nokia product family. In the product family, the features are typically developed independently and concurrently by different component factories. The component factories regularly release software components that are integrated in the products. In the integration phase, a set of features are combined together and presented in the user interface (UI) in a simple and coherent way. Some key challenges concern how to control the dependencies among the component factories, how to ensure the compatibility of the interfaces, and how to ensure that the architectural rules dictated by the reference architecture are enforced in the products. The reconstruction method helps the architects to extract the actual architectural configuration of the products and to examine the component dependencies at different levels of detail. The dependencies are represented with messages passed among the components and by C-like function calls within the components. With the component view the architects can look at the logical dependencies among the components and they can detect the clients and suppliers for each component. With the development view the architects can examine the source code organization and the include dependencies. With the task view, the architects can examine how the components are grouped in different OS tasks, and they can analyze the inter-task communication due to the exchange of messages. With the management view, it is possible to look at the organization of the component factories and their dependencies (caused by component usage among different factories). With the geographical view, it is possible to look at the geographical distribution of the components in the development sites. With the feature view, the architects can create Message Sequence Charts (MSC) showing the implementation of a set of features at the component level. The charts are based on the traces dynamically generated by the execution of the features on the target system. The reconstruction method supports the architectural concepts that are clearly specified in the reference architecture. In this way, the architects can automatically create the views from the implementation of the products, and they can look at and manage the assets from a high-level perspective. They can also validate their mental models with the concrete architecture.

6.2 Case study 2

We applied the reconstruction method for recovering the architecture of one Nokia mobile phone that now serves as a platform for other products. The first goal was to recover the current object oriented design of the system. With the tools we have extracted a model that contains the conventional C++ constructs (such as classes, methods, variables) and their dependencies (such as inheritances, method calls and variable accesses). We have organized the entities in logical groups according to the available design documents and according to the suggestions from the architects. The extracted model allows the architects to analyze the dependencies among the high-level logical subsystems and to examine the internal structure. The second goal was to detect the dependencies between the implementation and the external packages. We customized the extraction tools in order to detect the specific dependencies (such as asynchronous messages) that were requested by the architects. The final model allows the architects to navigate a complete architectural model starting from a high-level view of the system. The models also provided fresh architectural information for creating the platform of a new product family.

7. Conclusions

We have identified five approaches for creating a product family and the problems related to their evolution. In real product families the five approaches co-exist at the same time. We have proposed two methods for coping with the evolution of the family: architecture assessment and architecture reconstruction. The architecture assessment is typically applied during the grow phase of the product family in order to evaluate the new requirements and their impact on the product family architecture. The architecture reconstruction is typically applied during the consolidation phase in order to understand the actual implementation of the products, to integrate product features in the platform and to maintain the architectural integrity of the platform. Our case studies show that the two proposed methods have been beneficial for containing the risks implied by the evolution of the Nokia product families.

References

[1] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop, and A. Zaremski. Recommended best industrial practice for software architecture evaluation. *Technical Report CMU/SEI-96-TR-025, Software Engineering Institute, Pittsburgh*.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[3] G. Booch. Conducting a software architecture assessment. *Rational white paper, http://rational.com/products/whitepapers/391.jsp*.
[4] J. Bosch. *Design and Use of Software Architectures*. Addison Wesley, 2000.
[5] J. Bosch and P. Bengtsson. Component evolution in product line architectures. *Proceedings of the International Workshop on Component-Based Software Engineering*, 1999.
[6] E. J. Chikofsky and H. James Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, Jan. 1990. Definitions of a number of key notions in the field of reverse engineering are proposed. Forward and reverse engineering, redocumentation, design recovery, restructuring, and reengineering are described.
[7] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. Documenting software architectures: Views and beyond. 2003.
[8] D. Faust and C. Verhoef. Software product line migration and deployment. *Software Practice and Experience, to appear*, 2003.
[9] M. Jazayeri, F. van der Linden, and A. Ran. *Software Architecture for Product Families*. Addison Wesley, 2000.
[10] R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. *IEEE Software*, pages 47–55, November 1996.
[11] R. Kazman, M. Klein, and P. Clements. Atam: A method for architecture evaluation. *Technical Report CMU/SEI-2000-TR-004, Software Engineering Institute, Pittsburgh*, 2000.
[12] A. Maccari. Experiences in assessing product family software architecture for evolution. *International conference on Software Engineering (ICSE)*, May 2002.
[13] A. Maccari and C. Riva. Architectural evolution of legacy product families. *Fourth International Workshop on Product Family Engineering PFE-4*, pages 47–55, October 2001.
[14] C. Riva. Architecture reconstruction in practice. *Proceedings of the IFIP Working Conference on Software Architecture*, 2002.