JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2006; **18**:351–383
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.337

Practice

Continuous evolution through software architecture evaluation: a case study

Christian Del Rosso*,†

Nokia Research Center, Itämerenkatu 11–13, 00180 Helsinki, Finland

SUMMARY

The need for software architecture evaluation is based on the realization that software development, like all engineering disciplines, is a process of continuous modeling and refinement. Detecting architectural problems before the bulk of development work is done allows re-architecting activities to take place in due time, without having to rework what has already been done. At the same time, tuning activities allow software performance to be enhanced and maintained during the software lifetime. When dealing with product families, architectural evaluations have an even more crucial role: the evaluations are targeted to a set of common products. We have tried different approaches to software assessments with our mobile phone software, an embedded real-time software platform, which must support an increasingly large number of different product variants. In this paper, we present a case study and discuss the experiences gained with three different assessment techniques that we have worked on during the past five years. The assessment techniques presented include scenario-based software architecture assessment, software performance assessment and experience-based assessment. The various evaluation techniques are complementary and, when used together, constitute a tool which a software architect must be aware of in order to maintain and evolve a large software intensive system. Copyright © 2006 John Wiley & Sons, Ltd.

Received 9 September 2005; Revised 6 August 2006; Accepted 10 August 2006

KEY WORDS: software product family; software architecture assessments; scenario-based software architecture assessment; software performance assessment; experience-based software assessment

1. INTRODUCTION

Software architecture represents the earliest software design decisions. These design decisions are the most critical to get right and the most difficult to change downstream in the system development cycle. The software architecture is the first design artifact addressing reliability, modifiability, real-time performance, and inter-operability goals and requirements [1–3].

*Correspondence to: Christian Del Rosso, Nokia Research Center, Itämerenkatu 11–13, 00180 Helsinki, Finland.
†E-mail: christian.del-rosso@nokia.com

In 2000, the Computer Society approved the IEEE Standard 1471, which documents a consensus on good architecture practice [4]. In this standard, the architecture concept is defined as the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.

Designing the architecture and writing the software source code does not end the software life cycle. Software evolves and needs to be maintained in order to support new requirements and new hardware. It can be said that the major part of the work is done after the coding phase as described by Glass in [5] and by Pigoski in [6].

In large organizations such as Nokia, the job of the architect is particularly challenging: it involves continuous problem solving and maintenance interleaves with re-architecting activities in an effort to evolve the system functionality without breaking the architecture. Over time, architecture ages and weakens a system's capacity to incorporate new features; retrospectively, this means changing the architecture to make systems easier to expand and maintain. However, spending too much time on architectural changes that do not produce any direct benefit for the user should be avoided for obvious economical reasons. In this context, tuning and assessment activities allow a controlled evolutionary path for the software.

Architectural assessment is an activity of the architecting process that is targeted to evaluate the degree of fulfillment of quality, or non-functional requirements.

Architectural and re-architectural work is made more challenging when dealing with software product family architectures [7–10]. A software product family architecture is an architecture for a set of related products. Products in the family share common assets and architectural properties. Establishing product lines can bring several advantages: faster product derivation and higher software reuse may be achieved by porting a common architecture to support all envisioned products. However, assessments and tuning activities must consider the whole set of products, including the analysis of the tradeoffs, for the various products in the family.

Optimally, the architecture must scale up to support incorporation of future requirements. When the architecture cannot be extended any further, products can no longer be derived from the software product family causing higher development costs and the need to change the architecture. For these reasons, it is important to assess the evolvability of the architecture of a software product line during the software lifetime in a process of continuous assessment for architecture evolution. Currently, the literature on this issue is scarce, especially from the industrial realm.

This paper describes the experiences and the lessons learned from the architectural evaluation of large software systems in industrial settings. We have used three different assessment techniques to evaluate and improve over time one Nokia product line software architecture. The three software architecture assessment approaches used include scenario-based software performance assessment and experience-based assessment. The approaches are complementary and have been used to tune and assess the software product family architecture for evolution. When used together and included in the software lifecycle, the different assessment techniques bring several advantages for the evolution of large software systems.

The main findings and the experiment details have been presented accurately. However, for confidentiality reasons, a detailed description of the software architecture assessed had to be excluded from this paper.

The organization of the paper is as follows. Section 2 presents the notion of a software product family and introduces the challenges and issues of architectural assessments in this domain.

DOI: 10.1002/smr

In Sections 3, 4 and 5, the three assessment techniques used are presented. The first approach, described in Section 3, is scenario-based and scenarios are used to discuss and test architectural choices during brainstorming sessions. The second approach, in Section 4, is also a scenario-based approach, but it uses simulations and testing to evaluate and improve the architecture for quality attributes such as performance and dynamic memory usage. The third approach, in Section 5, is an experience-based assessment; interviews with key personnel are used to understand architectural problems that the architecture is facing due to the evolution and new requirements.

Each of these sections is structured as follows. First we illustrate the approach and its various steps. Then we present a case study and its results. The lessons learned present a set of key findings and lessons from the specific assessment method. A summary and discussion are given in Section 6. Also in Section 6, the methods are compared and the advantages of a continuous architecture assessment process are presented and discussed. Related work is presented in Section 7 and conclusions are given in Section 8.

2. EVALUATING SOFTWARE PRODUCT FAMILY ARCHITECTURES

A software product family is a set of software-intensive systems [4] having common assets and sharing architectural properties [7–10].

In a market where the demand for new and innovative products is increasing and where the complexity of software is never decreasing, having a product family architecture is a competitive advantage. The concept of a software product family, or a software product line, is derived from the hardware industry where new products were created by assembling and varying the existing parts of products of the same line, e.g., a car manufacturing company can have lines such as economic cars, luxurious cars, etc.

The products in the same family share the same reference architecture and part of the code. The software architecture for a product family must address the variabilities and commonalities of the entire set of products. New products in the family are instantiated from the same reference architecture. At the same time, variation points are specified to differentiate the products. The managed set of variation points guarantees a controlled diversification of the products that are part of the same family. A variation point is a location at which a change can occur in the software product line artifact [9] and examples of variability mechanisms are inheritance (if the component is a class in an object-oriented language) and configuration (by setting parameters at runtime or compile time) [8].

The study of the variabilities and commonalities of a product family is commonly referred to as domain analysis. Several methods and papers have been written on feature modeling and domain analysis. Feature modeling is part of the FODA (feature-oriented domain analysis) method [11], which is used in the requirements analysis for a product family, and FORM (feature-oriented reuse method) [12], where the method is aimed at software reuse. Other methods based on feature modeling are described in [13,14]. Managing variabilities is a challenging task. Experiences in managing infinite variability in mobile terminal software are described in [15].

Software reuse and a common reference architecture provide several advantages such as improved software quality and fast derivation of new products leading to time-to-market improvements.

The characteristics of the software product family architecture must be considered when assessing the quality attributes of software. Evaluating and tuning the quality attributes of a software product family has a broad scope that includes the whole set of products in the family, as well as the products' component parts.

DOI: 10.1002/smr

The evaluation of non-functional quality attributes includes a thorough analysis of the tradeoffs. A fulfillment of one quality attribute may have a negative impact on others. For example, improving the performance may affect the software maintainability and the software flexibility; fault tolerance usually comes at the expense of software performance.

An additional step in the analysis of the tradeoffs is represented by the software product family dimension. Re-architecting and refactoring the architecture implies the analysis of the tradeoffs for all of the products in the family. Improving the performance of a software product family implies enhancing the performance of the whole set of products or, at least, the products should not be penalized by the improvements made to a specific model.

Therefore, a deep understanding of the software product family domain is a prerequisite when evaluating this domain. However, different assessment techniques exist to evaluate software architectures and the software architect must use them appropriately. The assessment method can be chosen according to:

- the software quality attributes to be assessed;
- the input required by the assessment method and the architecture description format used;
- the process stage in the software lifecycle when the assessment is performed;
- the resources required and the time to be allotted to the assessment;
- the scope of the assessment, e.g., the entire software architecture, the multimedia system.

The software architect must be aware of the choices available and must use the right method appropriately at the right time knowing the strengths and weaknesses of each. Software development is a process of continuous modeling and refinement. Hence, different methods should be used in a continuous software architecture evolution through software architecture evaluation during the whole software lifecycle.

3. SCENARIO-BASED SOFTWARE ARCHITECTURE ASSESSMENT

Several architectural assessment practices and methods exist. Examples of scenario-based assessment methods are ATAM (architecture tradeoff analysis method) [16], SAAM (software architecture analysis method) [17] and the SBAR (scenario-based architecture re-engineering) [18] method.

We have used a scenario-based approach based on the methods mentioned above; however, we tuned our method to a software product family architecture. In addition, we tried to make the approach lightweight in order to be unintrusive and to not disrupt the normal software development activities.

The main advantage of this method is that it permits an evaluation of forthcoming changes and requirements even at the beginning of the feature's development and, therefore, before the full development work is done.

3.1. Method description

The scenario-based assessment method we use is made up of three different phases, see Figure 1.

The first step is the preliminary phase, where scenarios are gathered, classified and prioritized by the stakeholders. The second phase concerns the activities performed during the assessment day, a scenario walk through meeting where the architecture is evaluated according to the chosen scenarios. The final and third phase regards the activities performed after the assessment day, a follow-up phase, in which

DOI: 10.1002/smr

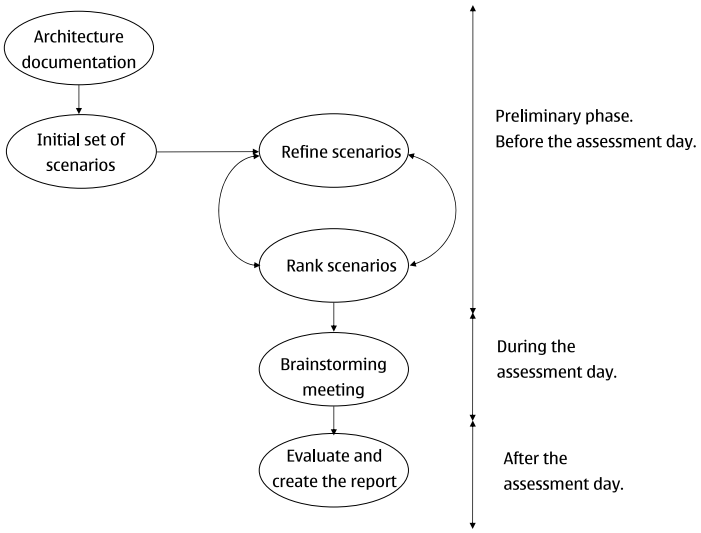

Figure 1. Assessment steps.

the results of the assessment are compiled in a report and the corresponding architectural improvement activities are started.

The method can be applied in the early phases of the software lifecycle at the beginning of the design phase. The architecture description is the required input for the assessment; however, the method does not require a particular documentation form or notation such as UML [19]. An informal architecture description such as Word documents and presentation slides can be used. A well-documented system, with updated documentation which uses a standard notation, greatly helps in the assessment process but is not always possible. This is especially the case with large software systems with many and ever-changing requirements. Software documentation is important and good links for further studies are given in [1,19–21].

In addition, for the success of the assessment, it is essential that all of the stakeholders possess a good understanding of the main architectural artifacts before the scenarios are walked through. A certain degree of knowledge of the architecture to be assessed is required. Basic knowledge of the protocols involved (e.g., General Packet Radio Service (GPRS) [22]), of the user interface architecture and of the behavior rules that dictate the behavioral paradigms of the software components are required.

In the following sections we describe the activities that take place during an assessment. We provide insights on how the various phases are carried out, a case study is presented in Section 3.2, while the lessons learned are highlighted in Section 3.3.

3.1.1. Preliminary phase

In the preliminary phase we define the list of scenarios. As main input we use the architecture documentation. However, we interview or contact the stakeholders involved in the architecture

DOI: 10.1002/smr

specification as an additional step. Features road mapping and requirement databases constitute an additional input for the scenarios creation phase.

As part of the preliminary phase, the assessment team organizes periodical meetings in order to inspect the collected scenarios and update the list according to business needs and new requirements.

The team of stakeholders taking part in the assessment consists of researchers, developers, architects and chief engineers. The stakeholders are asked to list what they believe to be the most important problems in the architecture in the context of the assessment's scope; evolvability, if the evolution is the focus of the assessment. The assessment team acts as a facilitator and handles the practicalities such as organizing the physical meeting with the stakeholders.

The scenario collection phase is iterative and scenarios are included, refined and ranked by the assessment team. In the refinement step, similar scenarios are grouped, irrelevant scenarios are discarded, and many others are re-elaborated. Still, there is a risk of leaving out some scenarios because of time constraints. Therefore, we prioritize the scenarios, producing a list with the most critical scenarios first.

Scenarios that contain features to be implemented immediately are regarded as having higher priority than those involving features due to appear in two years time. Similarly, scenarios that are thought to concern major architectural defects get higher priority than those which are believed to involve only limited changes in the source code.

Even though we have mainly used the scenario-based assessment method focusing on architecture evolution in which scenarios included future requirements, we may also want to include scenarios about current architecture. The reason why we have to include scenarios concerning current implementation is that they are usually felt to have been inadequately considered during architectural design.

The list of the scenarios are sent to the stakeholders for comments, thus re-iterating the scenario engineering process. We refine and discuss the scenarios repeatedly until the majority of stakeholders think we have found a good compromise between a sufficient level of detail and a manageable number of scenarios.

There is no general rule establishing when the scenario collection and engineering phase has been completed. We feel we can stop generating scenarios when the addition of new scenarios is no longer expected to reveal additional problems in the architecture.

The scenarios are then discussed in a brainstorming meeting on one or more days. The number of scenarios must be manageable. From our experience, a number of scenarios between 10 and 15 represents a good compromise for a brainstorming session of one day.

3.1.2. Activities that are performed during the assessment day

The assessment meeting is run in a brainstorming form. The stakeholders' team is made up of researchers, developers and architects. The participants are all technical people. According to their involvement and responsibility in the organization, they can take the following roles:

- chief architect for the software product family architecture;
- chief architect responsible for the part of the architecture considered;
- system experts, architects and developers;
- assessment facilitators; or
- secretary.

DOI: 10.1002/smr

At the beginning, the facilitators start illustrating the agenda and the participants briefly introduce themselves. Subsequently, the facilitators overview the assessment process detailing the activities to be performed during the day and on those to follow. To conclude the introductory phase, the software architect of the software to be assessed gives an overview of the architecture.

When the overview is completed, the scenario discussion phase commences. All attendees have already read the revised scenarios list. The assessment facilitators read and explain each scenario, raising discussion on a number of points. The stakeholders discuss the scenarios and reach an agreement on the technical conclusions, attempting to identify possible shortcomings.

In some cases, stakeholders may agree that a scenario can easily be fulfilled by the architecture. In other instances, stakeholders may find some weaknesses or unclear aspect in the architecture. During the brainstorming meeting, the secretaries keep minutes of the discussion. The assessment team will use the minutes as input for the reporting phase.

3.1.3. Activities that are performed after the assessment day

We collect and elaborate on the reports written by the secretaries. Different reports are eventually produced, for example, one highlighting the technical findings (i.e., the results of the assessment), the other describing the assessment process. The stakeholders inspect both reports, mainly focusing on the technical one. The facilitators produce a final version which is sent to the stakeholders for inspection.

The technical report highlights the identified defects. The defects generate a set of open items that will be used to generate action points and further activities targeted to solve the problems.

3.2. Case study

The main focus of the assessment was the evolvability of the architecture towards forthcoming requirements. Assessment of the network resource access control systems was one of the first case studies experimented on by our research group and has been described in [23].

The case study presented here is the assessment of the multimedia architecture and it was started to evaluate how the current architecture was affected by the forthcoming multimedia requirements.

Multimedia capabilities are now commonly used in every mobile phone; however, the software architecture was designed only for devices with voice call capabilities and requirements at that time did not contain any reference to multimedia features.

However, the architecture must support forthcoming requirements in order to evolve. Our task was to highlight potential flaws and to investigate the evolutionary path of the software.

In a multimedia environment, more sources of data exist. Within a device, data can be obtained, for example, from RAM, flash memory, from external storage such as multimedia cards or the local file system. Through a remote device, data can be accessed via GPRS [22] or WCDMA (Wideband Code Division Multiple Access) [24]. Via a local device we can access data via Bluetooth or Infrared and, in the future, we may have other methods and other data sources.

At the same time we have many different entities which access data; in the multimedia case, the main data consumers are various codecs and converters for video and audio. In the future there may be other video and audio formats.

The assessment had to focus on how the current architecture could evolve considering the new multimedia requirements. Some of the most important questions were addressed to understand and

DOI: 10.1002/smr

Table I. Example of a scenario in the multimedia assessment case study.

Scenario	Adding support for new data formats.
Description	We need to have an architecture that makes it easy to add support for new data formats.
Questions	How easy is it to add support for a new (video, audio, image) data format? What is the impact on the current architecture? What is the impact on applications when new data formats are added?

subsequently design how the entities in the system accessed the data, how the new multimedia functionalities were accessed by the application software, how the new data formats were incorporated into the sound and video system and how new multimedia (e.g., external and internal cameras) devices were handled by the architecture.

The software product family dimension introduced additional input for the assessment. The multimedia system had to be included in the software architecture; however, not all of the products had the same level of multimedia functionality support. High-end phones had to support advanced multimedia features while low-end phones had to have basic multimedia features. In general, low-end and high-end phones should not be affected negatively by the choices made in the product family architecture.

In the first iteration 15 scenarios were created, ranked and grouped according to various architectural issues such as features contention, synchronization between data sources and data consumers, and functional and non-functional requirements. During the second iteration only nine scenarios were selected for the assessment. The aim of the second iteration was to improve the focus and maintain a manageable list of scenarios while at the same time ensuring complete coverage of the assessment's scope. The iteration in the assessment process ended when additional scenarios were not found to add important topics for the assessment. The scenarios targeted the evolvability of the architecture and aimed at highlighting and stressing the current architecture considering multimedia requirements. An example of a scenario discussed in the assessment is in Table I.

The first version of the multimedia system architecture was under development but the design document was yet to be finalized. The assessment also served the scope by clarifying key issues before the bulk of the work was done.

The brainstorming meeting lasted one day and the 11 persons involved in the session had a stake in the architecture but not necessarily in the multimedia architecture. For example, the meeting included architects and technical experts for the networking aspect, experts in performance and persons responsible for the features contention architecture. There is no general rule on how to choose the stakeholders, but a balance must be found between choosing the important people and, on the other side, to have a number that is compatible for a discussion in a meeting in the same room.

We, as an assessment team, had the role of facilitators introducing the method at the beginning of the brainstorming session and then presenting the scenarios. In addition, we had the roles of secretaries.

The final assessment report was distributed for inspection and after few iterations was approved. The assessment report contained the technical findings and description of the assessment method. For each action point and open item we assigned a person responsible and a corresponding deadline.

DOI: 10.1002/smr

3.2.1. Results

Most of the scenarios discussed in the brainstorming session meeting did not highlight major flaws in the architecture. A major flaw in the architecture arises when a scenario expresses conflicting and contrasting design principles to the envisioned architecture.

However, seven scenarios raised open items and additional actions were considered. Some of the actions included further studies on the situations highlighted in the scenario. Two of the scenarios were instead supported by the architecture and did not involve re-architecting. In general, the assessment was a positive experience that permitted open discussion. One interesting outcome was an issue raised by one of the participants during the discussion which was included in one scenario and then on the action items list. The time for the assessment preparation was around 3 PM (person months) spent mainly by the assessment team to read the documentation and interact with the stakeholders.

3.3. Scenario-based software architecture assessment: lessons learned

As practice, we conduced a series of informal interviews with the stakeholders that participated in the assessments. During the interviews, we asked some questions about the assessment method. The questions were meant to clarify what benefits the assessment really brought (other than the obvious one: identifying architectural defects). Another research question concerned improvements to our (simple) assessment method targeted at achieving maximum benefit in an industrial setting, where time and financial investments matter. We elaborated the data with a critical approach trying to isolate problems in our method, isolating unexpected benefits of the assessment and extrapolating a list of lessons learned which could turn out to be useful in future instances of the experiments.

3.3.1. Documentation improvements

When we started the assessments, the architectural documentation in many cases was not mature. This was understandable since development of some parts of the system had happened recently and, documentation was still in the process of being written. Complete and updated documentation is difficult to achieve in the industrial realm considering the dynamic nature of the development process and the increasing number of new requirements. The assessments helped to highlight unclear parts of the documentation. In addition, the assessment permitted the updating of parts deemed important for the assessment's scope and, as a consequence, it improved the architecture design.

3.3.2. Enhanced communication

In some cases, the system under assessment was relatively new and its functionality was not clear to all of the stakeholders. The assessments that came in this context produced a better understanding of the systems. In addition, the assessments also highlighted the advantage of having all important stakeholders in the same room at the same time. The brainstorming meetings helped to raise discussion among people who work on the same project but in different areas and often at different physical sites. The fact that the assessments provided a good excuse for such people to gather and discuss technical issues was felt to be a positive thing, since the chances of that happening in the course of normal working life are slim.

DOI: 10.1002/smr

3.3.3. *Scenario collection and assessment focus*

For the assessment to be a success, the scope and goal of the assessment must be clear and explained to the stakeholders. A document stating the main reasons and the expected results of the assessment should be created and presented to the stakeholders.

Scenario collection is one of the most important phases in the assessment. This phase requires continuous interaction between the stakeholders and assessment facilitators. The stakeholders are the main contributors in the assessment and must be involved in the process.

The assessment team must be able to coordinate, including only those scenarios relevant to the decided scope and focus strictly on it. A too wide assessment scope would lose its effectiveness by giving vague results while a too narrow focus would not identify the main concerns.

Unfortunately, a magic rule for scenario selection and focus does not exist, however, the iterative steps and the ranking process provide guidance. Experience of the assessment team in the ranking process and the involvement of stakeholders is the key to success.

3.3.4. *Problem discovery during the whole software lifecycle*

The scenarios assessment method can be applied during the whole software lifecycle.

The advantage of using it at the beginning of development activities is given by the possibility to discover design problems before it is too late. When the software is in its early development stage it is still possible to make the appropriate decisions. However, when the bulk of work has been done, reverting the changes or noticing that the architecture is not appropriate for the evolution can be too costly.

However, the scenario-based software architecture assessment method can also be used to evaluate the current architecture. In many cases, the current status is overlooked and underestimating or overestimating existing or potential problems is common. Documentation and communication problems exist in every large software development activity. Such situations are more common when the R&D teams are distributed world-wide and involve thousands of people.

To reduce the risks associated with software development evolution, the scenario-based assessment method can be used to improve software architecture during the entire software lifecycle.

3.3.5. *Improvements for the software product family*

The assessment process described is not specific to software product families and it can be applied to any software architecture. However, we have been using the method to assess software product family architectures and we describe below the main benefits and lessons we have learned.

In the context of a software product family architecture, the assessment must consider the evolution for the whole set of products in the family. In scenario-based assessments each step in the process is affected by this dimension.

The scenario selection phase includes the analysis of the products in the family and how the features, optional or mandatory, are affected by the evaluation. In the case study presented, emphasis was placed on the impact of multimedia features to the whole set of products in the family, low end and high end.

The fact that the chief architect of the software product family was always included in our list of stakeholders ensured that the assessments took into consideration not only the specific software

DOI: 10.1002/smr

component but also the software product family architecture. In addition, during the whole process, the assessment team had to consider the software product family architecture dimension.

4. SOFTWARE PERFORMANCE ASSESSMENT

Software performance is a non-functional quality attribute that is not always considered at the design stage. The reasons are various. For example, it is not always possible to estimate the performance or the memory consumption of software, especially in the early stages of development and the normal attitude is to develop the system first and measure its performance later. In addition, the use of COTS (commercial off-the-shelf) components makes the job of the performance analyst harder since they usually do not contain any estimation of performance.

As a non-functional quality attribute, performance affects all of the components in a software product and it can only be analyzed by considering the software as a whole and by studying how the different components interact in order to achieve the desired target.

Mobile phones are embedded real-time systems. Embedded systems are constrained by limited memory, CPU and power. Their design is challenging since it requires efficient usage of available resources. Multimedia features are now included in the handsets and contribute to the stress of the system and software performance.

Moreover, the mobile phone as a real-time system has time constraints that must be respected. Software must not only perform fast but must respect hard real-time constraints. For example, failing to allocate a block of memory in time can cause a phone call to be dropped.

When the system is in the early development stage, software performance assessment can be done using mathematical models and simulation. Later in the software development phase, tuning activities can be used to improve software performance.

In the following sections, we first describe the method we have used to evaluate and improve the software performance of a software product family architecture during its lifecycle within Nokia. Then, a case study is reported. We will conclude the section on the case study with the results obtained. The presentation of the software performance method ends with the lessons learned section.

4.1. Method

We have used the software performance assessment method described here to discover the bottlenecks and potential problems in one Nokia software product family. The method is an adaptation of the Software Performance Engineering (SPE) method by Williams and Smith [25]. We have applied the method not only to evaluate and improve the performance of the software product family architecture in terms of CPU load and architecture design, but to also improve its dynamic memory management system [26–28].

The steps in the method are depicted in Figure 2. The method is scenario-based but unlike the scenario-based assessment methods presented in Section 3, it uses quantitative techniques to assess the architecture.

We have placed particular emphasis in the process on the software product family architecture dimension. Tuning a software product family means improving the performance of the whole set of products in the family and not affecting any of them negatively. Particular emphasis should be

DOI: 10.1002/smr

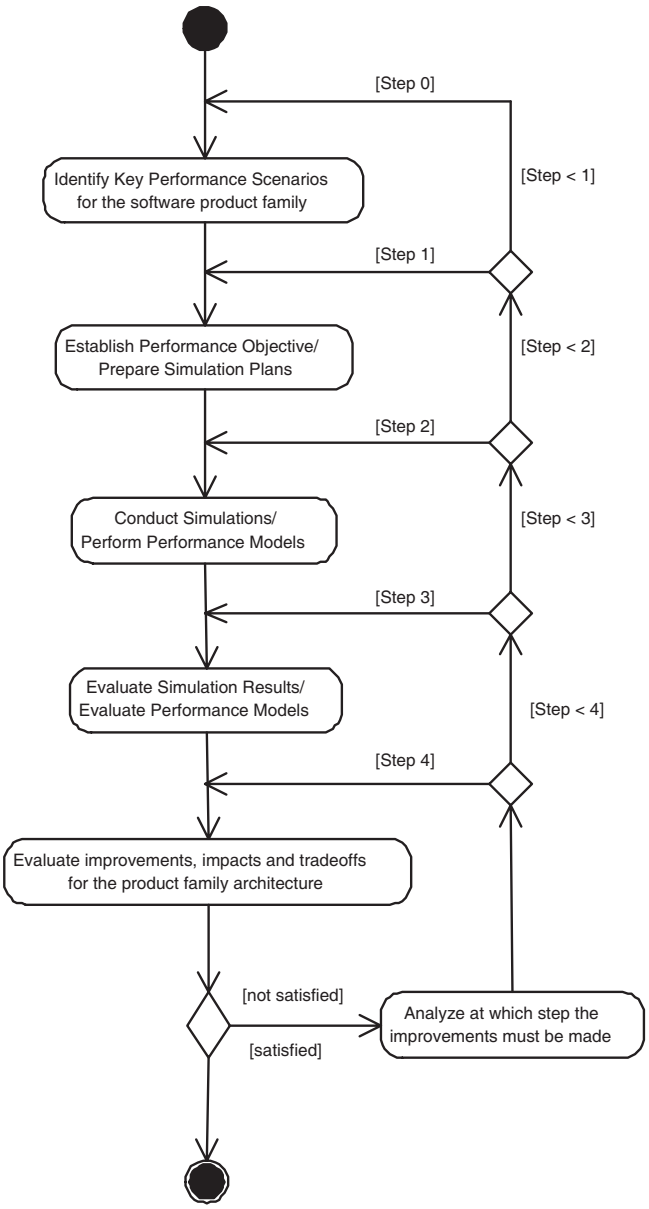

Figure 2. Software performance assessment steps.

DOI: 10.1002/smr

placed on tradeoff analysis considering that some features are only developed in a few products but that improvements can indirectly affect other features.

4.1.1. Identify key performance scenarios

The first step in the process is the assessment focus. The scenario selection phase identifies the assessment's scope. The architecture documentation and the stakeholders constitute the input in this phase.

The evaluation can be targeted to the product family reference architecture or to optional features instantiated only in a set of products of the family. We define a feature as a user-visible characteristic of the system, e.g., video call.

When focusing the evaluation on the product family reference architecture (the architecture common to all of the products), scenarios will include core set features. When the scope is on some optional features, the process will focus on the selection of challenging scenarios represented by those features. Multimedia features are examples of optional features and constitute an important topic to research since they are performance demanding.

A scenario describes the actions performed by combining one or more features. Key performance scenarios are scenarios that influence the systems the most in terms of performance and resource usage. Scenarios include user-visible features; therefore, features that are most frequently used by the user are on the list. Also included on the list of key performance scenarios are resource-intensive features (e.g., requiring large amounts of memory) even if these features are not commonly used.

In the context of real-time systems, we must consider qualities such as responsiveness. Features with strict real-time deadlines are also considered in our scenario selection phase.

The business importance of a feature can play a role in the process of scenario selection and the market research department can provide important input.

The iterative process of key performance scenario selection includes the stakeholders and the available architecture documentation. In the iterative phase we consider our focus, and select, rank and refine the list of scenarios.

The number of scenarios must be limited to avoid too wide a scope; however, too narrow a scope would limit the performance analysis. The number of scenarios to be defined also depends on the time frame the analysis has to be completed in. The experience of the performance analyst plays an important role in the process.

4.1.2. Establish performance objectives/prepare test plans

In this step, performance objectives and test plans must be defined.

Establishing objectives permits a clear path to the test plans. We must ensure that the objectives are defined using well-defined metrics and variables since *it is much easier to make measurements than to know exactly what to measure*.

The key performance scenarios defined in the previous step represent the input for defining the objectives. The representative variables and metrics to be extracted depend on the scenarios and the focus of the optimizations.

The variables to be analyzed concern resource usage, data characteristics, data path and time. For example, in the case of dynamic memory usage, our objectives were to optimize the usage

DOI: 10.1002/smr

of memory and at the same time improve software performance considering real-time deadlines. A definition of memory fragmentation was specified and time constraint metrics included the worst-case time performance for allocations and deallocations.

Based on the performance objectives and the stage that the software is in the lifecycle, we can define the test plans. Test plans include the specifications of the measurements tools and analysis techniques.

Several analysis techniques exist for analyzing the software architecture from a quantitative point of view. Instrumentation techniques can be used in later development stages. Prototyping, simulation and mathematical models can be used early in the software development lifecycle.

4.1.3. Conduct simulation, create performance models

Our method does not specify a quantitative method to be used and allows several techniques to be included in the process.

Once software has been developed, instrumentation techniques can be used to extract the events in the system and we can then study the software at runtime reconstructing the dynamic views of the architecture [29]. Instrumentation techniques have some drawbacks since they may affect software performance; however, wise use of the technique allows precise study of the runtime system [30,31].

Simulations of the entire system, or only a part of it, can be used to experiment on how the software behaves under different workloads or design alternatives. For example, in one project we simulated different memory management systems using trace instrumentation to extract memory allocations and deallocations executing key performance scenarios. Next, we built a simulation environment to test various dynamic memory management algorithms using real traces extracted from the running mobile phone software.

However, other quantitative methods can be included [32]. RMA (rate monotonic analysis) [33,34] can be used to estimate schedulability and to improve CPU load response time. Queuing models and Markov chains can be used in the early phases of a software development project to predict architecture performance and reliability. In the early phases, mathematical models and functions are used to create workloads based on the estimation of the use of key performance scenarios.

When software product family architectures are the object of the performance evaluation, we include representative members of the product family in both the analysis and modeling, basing simulations on their performance.

4.1.4. Evaluate simulation results/performance models

The evaluation of the simulations and performance models is based on the established performance objectives.

The improvements must be balanced with careful analysis of the tradeoffs. The analysis at architectural level can highlight performance anti-patterns and, as a consequence, the architecture must be re-factored [25,35,36]. Refactoring is the process in which the architecture design is transformed maintaining its semantics.

Design patterns are transformations that usually affect localized parts of the architecture. The best-known reference for design patterns is the work of Gamma *et al*. [37]. A useful reference for memory usage patterns in embedded system is represented by [38].

DOI: 10.1002/smr

In the case that the transformation affects the entire architecture, an architectural style can be imposed [8,39,40]. The pipe and filters and the layered style are examples of architectural styles.

Following an iterative process, once the bottlenecks and improvements are identified, we must ensure that the performance objectives are respected and verify their impact against other quality attributes. Quality attributes do not exist in isolation and the analysis of the tradeoffs includes the analysis of the performance gained against other software quality attributes. For example, security and fault-tolerance usually come at the expense of software performance.

Cost also has a stake in the process. Increasing the dynamic memory can decrease the fragmentation, but since the amount of memory that can be included in an embedded system is limited and also accounts for the final cost of a mobile phone, a balance must be found [41].

4.1.5. Evaluate the impact and tradeoffs for the software product family

An additional step in the evaluation is represented by the product family architecture dimension. The improvements must be analyzed against the impact on the whole set of products in the family.

The scope of the evaluation can be targeted to the whole software product family architecture. In this case, key performance scenarios will include core features which are features of the common reference architecture. However, new forthcoming requirements can affect only a subset of products in the family, e.g., multimedia features. Scenarios in this case will focus on the specific and performance challenging features. However, in both cases the impact and tradeoffs must be thoroughly analyzed.

Modifications to the reference architecture can have negative implications for some products. *Vice versa*, a change to a product architecture can require an architecture transformation back to the reference architecture thus affecting other software components.

The impact analysis follows the features dependency model [11], and the static [42] and the dynamic views [29] of the architecture.

4.1.6. Iterations

The method is iterative. Iterations serve the scope of refining some of the steps in case we have not reached the objectives or we want to improve and analyze the software in greater depth.

The iterations can start at any previous step in the process, however, a sequential path must be followed from that step. For example, if new scenarios are added we must run all of the simulations and analyses again.

4.2. Case study

We have used the performance assessment method to evaluate and optimize one Nokia product family architecture. We have tested our method in two different experiments. The scope of the first experiment was to tune and optimize the software architecture for performance [26].

In another project we used the method to evaluate the dynamic memory management system of the software product family by simulating various dynamic memory management systems and analyzing their performance and memory usage [27].

The case study described in this paper regards the first experiment, performance tuning of the software product family architecture.

DOI: 10.1002/smr

Our task, as the assessment team, was to analyze and tune the software product family architecture with the scope of finding bottlenecks and hot spots. In addition, the evaluation had to assess the architecture for evolution analyzing how the current architecture was affected by the new requirements.

The architecture documentation of the software product family was the input for the assessment. Key performance scenarios were selected using current documentation and through informal discussion and interaction with the main stakeholders.

At the end of the scenario selection phase, we selected the most used features in the mobile phone common to all of the products in the product family as key significant scenarios. We wanted to focus our assessment on the significant core set of features. Some of the scenarios selected for the study were:

- phone start-up;
- scrolling the phone book;
- incoming call.

In order to provide a clear description of the use cases, we described the scenarios following the template and advice of Cockburn given in his book *Writing Effective Use Cases* [43]. The phone turned off was the initial condition for all scenarios as well as the final condition. For example, the *scrolling the phone book* scenario represented the execution of phone start-up, opening of the phone book followed by the action of scrolling the phone book a defined number of times. Additional information was also provided. In the scrolling the phonebook scenario, the number of contacts in the phone book and whether the contacts are stored on the phone SIM card or the phone memory are important details.

The system considered used message passing communication as the communication mechanism between the processes. Therefore, the message behavior and message performance were the targets of our studies.

In the test plan, several architectural views were investigated. In the end, five architectural views were selected for study. The CPU load view, the MSC (messages sequence charts) view, the message statistics view, the runtime coupling view and the static view. Figure 3 shows the architectural views used.

The CPU load views showed the load of the processes while running the key performance scenarios. The MSC views showed the message interactions of the processes during the scenarios with a finer level of granularity than the CPU load view. The messages statistics view presented the CPU load time spent to serve different message types. The runtime coupling view represented the runtime connections of the processes in a graph format and the weights in the arcs were the number of messages exchanged. Finally, the static view was used to understand the source code software dependencies (modules and function calls dependencies). For further discussion on the architectural views, their use and synchronization, see [29].

We conducted the simulations and reconstructed the different views using trace instrumentation. Traces from the running phone were extracted using trace instrumentation inserted into the software source code. The traces extracted were stored in text log files and we reconstructed the different architectural views using tools that we developed.

Two mobile phones with different capabilities (the high-end and low-end part of the product family) were used to analyze the performance in the software product family.

From the architecture reconstruction step, the evaluation and the analysis steps followed.

DOI: 10.1002/smr

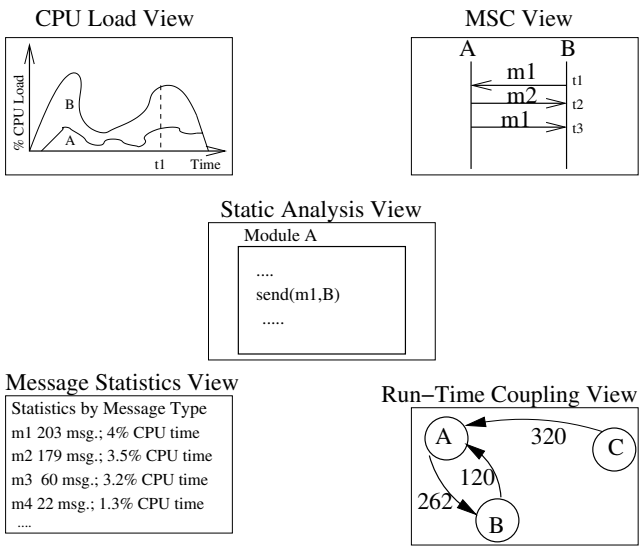

Figure 3. Architectural views reconstructed.

4.2.1. Results

We reconstructed different architectural views from traces extracted from the running mobile phone software.

Through analysis of the views and interaction of the different views we obtained interesting results.

The runtime coupling view revealed the *GOD class* anti-pattern [25]. Looking for anti-patterns was one of our targets and the GOD class anti-pattern was easy to find. A process was performing most of the work and other processes were accessing and sending message requests to that single process. The study highlighted a poor distribution of the intelligence in the architecture and a bottleneck represented by the GOD process.

The optimizations did not only concern the architectural design. We discovered a legacy part in the source code that was using resources for sending messages conforming to an outdated version of the software architecture. That part of the source code was executed at phone start-up; it did not halt the system but it spent CPU time. Without our cooperative study of the views it would have been difficult to spot.

4.3. Software performance assessment: lessons learned

The lessons we learned came from using our method for assessing and evaluating the software performance and memory management system of one Nokia software product family architecture.

DOI: 10.1002/smr

4.3.1. *Evaluation team not part of the development unit*

The evaluation team in our case came from the corporate research center which is a separate unit not from the business development unit.

A quantitative approach such as performance assessment is less affected by biased opinions than qualitative assessments. However, an external team can bring a different perspective and new design ideas. A development team performing its internal performance assessment may overlook some design decisions or take them as optimal or granted while an external team can inspect the system from a neutral point of view.

However, a weakness in having an external assessment team is represented by a lower knowledge of the system and, subsequently, by the time that must be spent in learning the characteristics of the new architecture.

4.3.2. *The focus must be well defined*

As in every evaluation, the focus must be well defined before conducting the experiments and simulations.

Key performance scenarios define the scope and are the starting point for the analysis. A well focused analysis is the basis for the assessment and the objectives and test plans should be well specified and agreed on with the persons commissioning the work.

When the focus is memory usage, scenarios that affect and stress the memory will be the center of interest. When CPU load is the target of the evaluation, scenarios with CPU-intensive features will be selected.

However, since we are concentrating the evaluation on user-visible features, we will need to understand feature usage patterns. Frequently used features impact on the everyday lives of users. Features that are not frequently used but when used require a lot of resources should be included in the scenarios. The business importance of the features also plays a role in the key scenario performance selection.

The features to be included can be numerous, however, only with a specific focus and efficient ranking activity is it possible to have a manageable list of key performance scenarios. Moreover, in the process, the experience of the assessment team plays a role.

4.3.3. *Documentation improvements*

The main target of the performance assessment is to find bottlenecks, hot spots and to assess the architecture for evolution.

However, an indirect benefit of these evaluations is documentation improvement.

Software performance evaluation includes the dynamic view reconstruction of the architecture. For large software intensive systems with fast evolution lifecycles it is difficult to maintain the documentation updated to the actual architecture.

As part of the assessment activities, we have found the architecture reconstruction activity to be beneficial. The architecture consistency check helps in updating documentation and spotting legacy code. At the same time, potential problems and inefficiencies can be discovered by identifying when architectural rules have been broken.

DOI: 10.1002/smr

4.3.4. *Performance improvements at different abstraction levels*

During our performance assessments we have applied architecture transformations but, at the same time, source-level optimizations have been considered.

At the architectural level we focused on software anti-patterns. Anti-patterns document common mistakes made during software design and, consequently, architecture transformations are used following the best architecture design practices: software architecture design patterns.

Design patterns condense the knowledge of experienced architects when dealing with common design choices in software design [35,37,44]. When software transformations affect the whole software architecture, architectural styles can be applied [8,39,40]. Example of performance patterns and anti-patterns can be found in the work of Williams and Smith [25,36].

While performance improvements made at the level of architecture design have a great impact on software performance, we should not discard lower-level details and implementation issues at the source-code level. Algorithms and coding details influence software performance and responsiveness. A quick-sort is more efficient than a bubble-sort and a hash table search is faster than a sequential search.

Architecture transformation and coding refactoring are part of the evaluation process. However, tradeoff analysis in also included in the evaluation process.

The analysis of tradeoffs include a thorough investigation of the cost of implementing the changes versus the benefits obtained. Moreover, the tradeoffs are between quality attributes. Performance can be penalized by software maintainability and flexibility.

For example, the blackboard style [39] is less suitable where performance is an issue since it introduces overhead while forcing communications via the blackboard. However, the style introduces flexibility to add additional components without affecting existing components.

4.3.5. *Improvements for the software product family*

Software product family evolution adds an additional step in the software performance assessment method: analysis of the evolution of a set of related products with commonalities and variabilities.

New demanding features can be implemented and instantiated in certain products. The software architecture must be able to support the range of products with different requirements in terms of performance and memory.

In the case of mobile phones, multimedia features are particularly demanding in terms of performance. For example, video calls require strict time deadlines, an increasing CPU processing speed and memory consumption. Not all of the mobile phones in the portfolio will support such features; however, the architecture of the software product family must be able to handle such variabilities.

Key performance scenarios will include core features when the assessment is targeted to the software product family. When a subset of optional features is the scope of the assessment, key performance scenarios will include the features that will be instantiated in the subset of the products of the family.

The tradeoffs analysis, however, will always include the whole set of products. Architecture transformations for the software product family include analysis of the tradeoffs of the optimizations against the whole set of members of the family.

DOI: 10.1002/smr

Increasing the commonalities in the family architecture follows the normal evolution of the software. New features will be implemented in high-end phones and once commoditized, they will be moved into the core set features part of the family architecture. As Lehman states [45], the functionality has a tendency to move from the perimeter of the product towards the center, as the innermost layers extend and support new functionality.

An increase in commonalities also means exploiting the benefits of software product family architectures with improvements in software maintainability, software quality and time to market.

On the other hand, additional variability can be introduced to allow more flexibility in the architecture. For example, different dynamic memory management systems can be introduced to serve products with different memory requirements. The tradeoffs in this case are between flexibility and maintainability. In addition, various levels of variability can be included in the architecture (e.g., runtime binding and static binding) and must be evaluated.

In some cases, the evaluation may highlight a deeper divergence in the product family architecture with conflicting requirements within the family. The divergence can be due to new requirements and architectural choices that conflict with a subset of the products or to new products created to satisfy new requirements.

Several scenarios are possible. A split in the product family architecture can happen with the creation of a new branch through software product family cloning. In this case, the new branch will follow its own evolutionary path. A less drastic scenario is the creation of a new branch part of the same product family, but this time the branch is part of a hierarchical tree of the product family, what Bosch [8] describes as specialization. In this second case, the commonalities of the architecture are still preserved and the evolution regards the product family tree with its own specialization branches.

5. EXPERIENCE-BASED SOFTWARE ARCHITECTURE ASSESSMENT

Experience-based software architecture assessment bases its strengths on the experience of the stakeholders involved in the software development process. These stakeholders are the chief architects, architects, developers, requirements engineers, and all of the persons involved in the software development activity.

Bosch [8] describes experience-based assessments as a way to evaluate and assess a software architecture. We have employed this concept and gained experiences in industrial case studies. The outcome has been valuable within our organization and in this paper we have included the lessons learned.

The ability to understand the current problems of the architecture and the potential problems concerning future requirements is quite challenging when large software systems are considered, especially when the development teams are distributed geographically in different time zones and with cultural differences.

The method can be applied to any large software intensive system for the evaluation of the software architecture and its evolution. Experience-based assessments can be performed during the whole software lifecycle and the introduction of the practice has advantages that will be described in this paper.

Even though the assessment is based on interviews with the stakeholders, its outcome is technical and not subjective. People tend to have strong opinions and preferences regarding software design,

DOI: 10.1002/smr

software tools, etc. The assessment team has the job of cleaning and refining the outcome from the pure beliefs, tackling and analyzing only rational motives. As part of the analysis, subsequent quantitative assessment activities can be initiated from the list of open issues.

5.1. Method

Defining the scope of the assessment and listing the main stakeholders is the first step in the process. The second step is the interviewing phase in which stakeholders are interviewed by the assessment team. The analysis step is the last phase where material collected during the assessment and architecture documentation are analyzed and evaluated by the assessment team. Iteration is in the process since new people will be added during the interviewing phase and analysis and refinement work is continuous until the end of the assessment.

5.1.1. Defining the scope of the assessment

The experience-based assessment can serve several purposes, for example to understand and evaluate the software development process in a company, or it can be purely technical. In a case in which the focus is purely technical, the method is used to assess the software architecture for its current status and evolution. In both cases, the focus must be specified and defined with the representatives commissioning the work.

In the initial phase, the architecture documentation is collected. However, the architecture documentation will be collected during the whole process when, during the iterations and the interviews, more specific knowledge of the software architecture will be needed.

5.1.2. Selecting the stakeholders

After defining the scope of the assessment, we must define the list of people to interview. The chief architect is the person with a high-level view of the software development process and is able to decide on the initial list of persons to interview.

The list is fundamental to start but is not definitive. During the interviews, new persons will be added to the list when the need to investigate further topics emerges. The same interviewees will point out the persons with knowledge of specific topics or parts of the software architecture.

We divide the stakeholders into three broad categories to be interviewed. Requirement engineers, who constitute the first category, are the persons responsible for collecting forthcoming requirements and evaluating their potential impact on the architecture. The second category is represented by the architects, who are the persons responsible for the design of some part of the software product family architecture. Finally, the third category is represented by developers and experts of some key quality attribute, e.g., performance. The categories must be structured in order to cover the stakeholders within the scope of the assessment. In the case that a different organization structure is in place, the assessment team must evaluate it.

The interviews are organized by interviewing all of the people in the same category and then continuing to the next category. However, new people are continually added to the list and when the development team is distributed geographically, the persons at the same site are interviewed together for practical reasons.

DOI: 10.1002/smr

5.1.3. *Making the interviews*

The assessment's scope and methodology is presented to the stakeholders being interviewed. The comments of the stakeholders can contribute additional information towards refining the scope and improving the assessment method.

The interviews are semi-structured and the interviewees are free to discuss their main concerns about the architecture from their perspective. However, the assessment team must be able to guide the interviewees within the focus of the assessment.

The assessment team must take the minutes of the interviews and it must be clearly stated that the final report will not contain any reference to the names of the persons being interviewed. In the end, only technical findings will be reported to management. The statement is important; the interviewees must be free to state their opinions without fear of superiors.

5.1.4. *Analysis and final report*

The analysis is done as the interviews proceed. The interview material is revised and analyzed every time. New documentation is collected during the assessment. In case no documentation is available, the assessment outcome must highlight the fact for improvements.

The interviews, even though they represent the stakeholders point of view, must contain some evidence of the claims and must be grounded technically. Opinions can be biased towards a particular design choice while the role of the assessment team during the interviews is to guide and inquire about references and actual facts.

In the analysis work, only technical findings are evaluated and pure opinions will be discarded. In some cases, opinions can highlight the need for a more accurate analysis. After all, software development is still a human activity; great designers create great designs and experience is an important virtue.

The open items are highlighted and further research projects are part of follow-up activities included in the assessment outcome. Quantitative assessments and further studies are the natural output for this kind of assessment.

The report will be structured by topic and focused on the assessment's scope: only topics relevant to the assessment must be included. In addition, the report will highlight the open items and action points. The report will be delivered to the stakeholders for feedback. Comments and corrections are possible in this phase.

The final outcome is delivered to the person founding the project. The list of open items and action points is handled by the persons in the management units with the assessment team. Action points and open issues are prioritized and ranked. Every action point will be assigned a responsible person and a fixed deadline.

5.2. Case study

The scope of the assessment included one of Nokia's mobile terminal software product families. The experiment was first described in [46].

The experience-based assessment was focused on the whole software product family architecture. Multimedia requirements, communication protocols and the support of new hardware for the whole range of products in the family constituted the challenges for the architecture evolution.

DOI: 10.1002/smr

The assessment had to assess the architecture for evolution and understand the impact of new requirements on the current architecture. The assessment was focused on the software product family architecture, not to a subset of its products, e.g., multimedia phones.

During the first phase, the chief architect who commissioned the work listed 18 persons with their title and tasks within the organization. Those 18 persons were to be interviewed first. The persons on the list were ranked given the large number to be interviewed and the limited time for the project.

However, during the whole process, the list of interviewees grew to a total of 35 stakeholders with eight different development sites covered in four different countries.

The increase of stakeholders and interviewees was natural. As the work proceeded, there was a need for an in-depth investigation of certain parts of the architecture and the persons interviewed suggested additional parts.

The semi-structured interviews revealed, contrary to belief, a well-structured assessment. Recurring topics and concerns were raised during the interviews. However, biased comments appeared; as matter of fact, for an experience-based assessment we found negative and positive attitudes in the stakeholders.

Documentation was collected during the analysis and we, as the assessment team, had to study the documentation, ask for missing documents and, in some cases, ask the stakeholders directly.

The report was a document structured by topic, from operating system concerns to the section dedicated to wish lists: the list of features and mechanisms that stakeholders wanted in the architecture. The action points were ranked and the estimated impact of the changes was included. The work was done with the cooperation of the stakeholders, the assessment team and the chief architect.

5.2.1. Results

The case study was an experience-based software assessment. The assessment lasted one year. We interviewed 35 people at eight different development sites distributed in four different countries.

The outcome of the assessment was the assessment report, a document of 70 pages including the main findings. We, the assessment team, were three researchers of the Nokia corporate research center, and we had the job of organizing, scheduling and conducting the interviews, refining and ranking the topics and, finally, releasing the main findings.

The importance of the assessment outcome was high. Communication is one of the main problems in large organizations and our work permitted us not only to highlight several problems in the architecture but also to deliver a comprehensive document on the architecture evolution. As an example of a communication problem, in one case, some stakeholders raised a problem that had been fixed months before.

With the report we presented the status of the current architecture. The report constituted a snapshot of the architecture status and its path for evolution.

From the technical perspective, we had 65 action points. The description of each action point contained a problem, a solution and its business importance. Tradeoffs analysis had to balance the problem with the business importance and the amount of resources needed to address that particular problem. The collaboration of the stakeholders, the chief architect and the assessment team was essential.

5.3. Experience-based software architecture assessment: 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 the form of advice.

DOI: 10.1002/smr

5.3.1. *Well-focused scope*

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 clear idea of the entire process and of the practical impact and value of their contribution.

The structure of the interviews forced the stakeholders to concentrate on very specific issues. We built the interviews on information we had gathered during previous interviews and had gradually refined. In many instances, the stakeholders repeatedly quoted a certain issue as a problem which increased the validity of their claims.

5.3.2. *Communication vehicle*

The main purpose of the assessment was to identify and highlight possible weaknesses of the product family architecture. However, almost all stakeholders rated the role of the assessment as a communication vehicle as very important. This is consistent with previous research work on the subject, but it was particularly evident in our case study where the size of the architecture was fairly large and complex. In addition, the geographical and organizational distribution of the development and maintenance departments naturally hampered communication.

In more than one instance we came across a stakeholder who quoted a problem that had already been fixed elsewhere, evidently without proper communication. The assessment partially helped to overcome this problem. Moreover, the report we produced served as a sort of 'white paper' on the status of the product family architecture and the main problems it suffered from.

5.3.3. *Assessment team not part of the development team*

An internal assessment team, part of the organization that is responsible for developing the architecture, is naturally subjected to more bias and influence. The reasons can be various, from trusting your close colleagues statements more than others, to having a vested interest in promoting a positive evaluation of the architecture. All of these situations must be avoided during the assessments and an external evaluation team can provide the solution.

5.3.4. *Assessment must be based on technical reasoning*

People tend to have subjective answers to technical questions. During the course of our interviews, we noticed that some stakeholders tended to minimize the relevance of the problems, assuming an 'everything can be solved' attitude. Others, however, were more pessimistic, implying that 'the whole company will suffer from this if we do not do something soon'. As always, the truth lies somewhere in the middle. The assessment coordinators must try to keep a balance and report facts as they are, with as little emotional bias as possible. Reasoning must adhere to technical facts and in cases where clear evidence is not found and the problem is considered relevant, subsequent follow-up activities can be started as part of the assessment process.

DOI: 10.1002/smr

6. CONTINUOUS SOFTWARE ARCHITECTURE EVALUATION

In this paper we have presented the experiences gathered using three different assessment methods. Each experience has been described using the following structure: first we have illustrated the different steps of the method used, then we have presented a case study and its results. In the lesson learned section we have given advice and explained what we have learned from applying the method.

The methods presented are a scenario-based software architecture assessment, a software performance assessment and an experience-based software architecture assessment. The three methods represent an assessment framework that we have used to evaluate and assess one Nokia product family architecture during its software lifecycle. The proposed methods have been beneficial for containing the risks implied by the evolution of the Nokia software product family architecture.

In the following three sections we summarize the characteristics of the three methods. Then we present the benefits of the assessments in a continuous software architecture evaluation process during the software lifecycle.

6.1. Characterization of scenario-based assessment

The scenario-based software architecture assessment method is suitable for evaluating and assessing the architecture for modifiability and evolvability; however, other quality attributes can also be assessed.

In general, the method can be applied during the whole software lifecycle, but its main benefits come when it is used to evaluate the impact of new design decisions and before the software is implemented. Changes made at that stage in the software lifecycle are less costly and mistakes can be recovered easily.

No particular form of the documentation is required as input although documenting the software architecture using a standard notation is a good practice [20]. On the other hand, at the beginning of the software design stages, the documentation is incomplete or missing and the method can still be used. A side benefit of the assessment is also the improvement of the software documentation.

The scenario-based assessment method presented constitutes a lightweight approach. A heavyweight approach may negatively influence the strict software development activity schedules and budgets and overly sophisticated methods may not pay off.

The main event is represented by the brainstorming session, usually lasting one day with the participation of all of the stakeholders. The work of evaluation and analysis is performed afterwards by the assessment team.

The need to have the brainstorming session with all of the stakeholders on the same day makes the assessment method suitable for evaluations where the number of persons does not exceed the meeting room size ensuring smooth communication.

The assessment method is not specifically targeted to software product family architectures. However, we have described the enhancements, and presented the case study and the lessons learned for software product family architectures.

6.2. Characterization of software performance assessment

The software performance assessment method is primarily targeted at evaluating the performance and the memory efficiency of a software system. However, other quality attributes that require quantified measurements can benefit from the method.

DOI: 10.1002/smr

New features demand increased performance and a software product that is not able to perform adequately is soon out of the market. Therefore, software evolvability is also assessed. The method helps identify potential architectural problems and aids the architecture development activities for software evolution.

Architecture documentation and its dynamic view is the main input of the method. In some cases, the software source code can be necessary, for example when source code instrumentation is used and when source code level refactoring and optimizations are done.

The method can be used during different stages in the software lifecycle. In the early stages, modeling, simulations and prototyping can be used. In later stages, performance tuning with trace instrumentations and measurement techniques can be used. The specific technique used and the scope of the assessment determine the resources needed to perform the assessment.

Performance depends on the architecture design and on implementation details. Both optimizations are possible using the performance assessment method.

We have used the software performance assessment method to evaluate software product family architectures. Our main emphasis and contribution is on the discussion and on the evaluation of these architectures, extending methods such as the SPE method [25].

6.3. Characterization of experience-based assessments

Assessing the suitability of the current software architecture and its evolution is the main objective of the method.

The experienced-based assessment method gains its strengths from the experience of the stakeholders involved in the process. The main input of the process is the material collected by interviewing selected stakeholders and the architecture documentation.

The resources needed for evaluating the architecture are based on the number of stakeholders participating in the assessment and on the assessment's scope. In general, the approach is medium weight in terms of time spent. For example, two hours per interview can be allocated for each stakeholder. The analysis phase, performed by the assessment team, consists of analyzing the material from the interviews, studying the software architecture documentation and eventually by adding and reviewing additional documentation.

Large software-intensive systems with a wide assessment scope are primarily the target of the experience-based assessment method. The approach allows the evaluation of the *entire architecture* for evolution as presented in the case study.

Software product family architectures are large software-intensive systems, in particular, they are a set of such systems. We have applied the method by evaluating the suitability of the software product family for evolution. The correct evaluation was done by considering the evolution of the software product family architecture and by studying the evolutionary path of the products part of the family.

6.4. The three methods together

We have used the three methods presented in this paper to assess and support the evolution of one Nokia software product family architectures during its lifecycle. Table II summarizes and compares the characteristics of the three assessment methods. The table structure is taken from [47] and we added the last two rows.

DOI: 10.1002/smr

Table II. The methods compared.

	Scenario-based software architecture assessment	Software performance assessment	Experience-based software architecture assessment
Quality attributes covered	Mainly modifiability and evolvability; however, other quality attributes can be assessed.	Primarily performance, memory efficiency and evolvability. The quality attributes are quantified.	Suitability of the current and future architecture design.
Object(s) analyzed	Architecture documentation in no particular form.	Architecture documentation, mainly the architecture dynamic view and optionally the software source code.	Architecture documentation in no particular form.
Project stage when applied	During the whole software lifecycle, preferably in the early software development stages or when important changes are planned.	During the whole software lifecycle. Modeling, simulation and prototyping can be used in the early stages; measurements in the later stages.	During the whole software lifecycle. Depending on the scope, it can be done during or after the architecture design.
Approach(es) used	Scenario-based walkthrough that takes place during a brainstorming session with selected stakeholders.	Scenario-based with simulation modeling and prototyping, depending on the software stage in the lifecycle.	Interviews with selected stakeholders.
Resources required	Lightweight assessment approach method. The lightest of the three.	Depending on the specific evaluation technique used, medium to high evaluation approach.	Medium-weighted approach. The resources required depend on the assessment's scope and on the number of stakeholders involved.
Additional notes	This method has proved to be effective when the stakeholders do not exceed the reasonable size for a meeting in a room, so that communication during the meeting can happen smoothly.	The method has proved to be effective in improving the performance at different architecture abstraction levels: architecture design and source code level.	The method has been successfully used in large software-intensive systems with a wide assessment scope such as the assessment of the *whole architecture*.
Software product family architecture compatibility	Not targeted specifically for software product family architectures, but it has been validated in industrial case studies.	Enhanced from the SPE method to include software product family architectures.	Method targeted to large software systems, therefore it is valid for software product family architectures.

DOI: 10.1002/smr

Table III. Lessons learned.

	Lessons learned
Scenario-based assessment	Documentation improvement, Section 3.3.1. Enhanced communication, Section 3.3.2. Scenarios collection and assessment focus, Section 3.3.3. Problem discovering during the whole software lifecycle, Section 3.3.4. Improvements for the Software Product Family, Section 3.3.5.
Software performance assessment	Evaluation team not part of the development unit, Section 4.3.1. The focus must be well defined, Section 4.3.2. Documentation improvements, Section 4.3.3. Performance improvements at different abstraction levels, Section 4.3.4. Improvements for the software product family, Section 4.3.5.
Experience-based assessment	Scope well focused, Section 5.3.1. Communication vehicle, Section 5.3.2. Assessment team not part of the development team, Section 5.3.3. Assessment must be based on technical reasoning, Section 5.3.4.

Each assessment method has its own characteristics. The choice of which assessment method to use must be based on the specific quality attributes to be assessed, on the input documentation required, the stage of the software in the lifecycle and the resources required. Table II is the result of our experiences and the 'Additional notes' row in the table highlights important findings of our case studies.

By appropriately using the methods and by including the assessments practice in the software development activity, we believe that the software evolution process can be greatly improved.

The lessons learned represent an additional contribution of our work and Table III summarizes the lessons learned from our case studies. In this paper we have presented an in-depth discussion of the lessons learned. Table III reports references to their section numbers.

In addition, as we noticed in all of the experiences, an important benefit of the assessments, which cannot be properly classified as a goal, is the improvement of the communication between stakeholders, architects and developers and the identification of defects or unclear parts in the architectural documentation.

Communication problems are one of the biggest problems in software development activities, especially in large software-intensive systems with development activities distributed geographically all over the world. A continuous assessment process included in the software development lifecycle can mitigate the risks for architecture evolution and at the same improve the communication within the organization.

7. RELATED WORK

Software architecture assessments are performed to evaluate how the system fulfills quality attributes. Modifiability, evolvability, security and performance are called non-functional quality attributes and

DOI: 10.1002/smr

represent important properties that the software should have. The IEEE standard 1061 provides a definition of software quality [48].

Depending on what quality attribute is evaluated, qualitative or quantitative methods can be applied. Regardless, the methods are not mutually exclusive and can be complemented. Qualitative methods can be based on scenarios, checklists, questionnaires, or can be experience based and are used, for example, to estimate modifiability and evolvability [49].

A good survey on software architecture analysis methods is the paper by Dobrica and Niemelä [50]. The study focuses on scenario-based architectural assessment methods and compares them in order to find similarities, differences and aims to offer guidelines on how to choose the appropriate method in different circumstances.

The Software Engineering Institute (SEI) has created some well-known scenario assessment methods: the SAAM [17] and ATAM [16,51]. The work on the assessments of SEI has been summarized in a book written by Clements *et al*. [47].

Another important and well-known scenario-based method is presented by Bengtsson and Bosch [18]. In their work a particular stress is given to architectural (re)design.

We have learned from previous scenario-based assessment methods and have used a light version approach. Maccari [23] reports the experiences of two case studies using our scenario-based assessment method.

A fundamental work on software performance is represented by Smith and Williams with the SPE method [25,52]. In addition, Williams and Smith proposed PASA^{SM} [53], an architectural assessment method with scenarios specifically targeting software performance.

We have enhanced and elaborated the SPE to include software product family architectures, especially in the scenario definition phase and analysis phase.

The software performance assessment uses quantitative techniques for evaluating quality attributes. For example, for the estimation of worst-case response time in real-time systems, RMA [33,34] is used. In addition, queuing models for performance prediction and Markov chains for reliability reasoning can be used when appropriate [32].

In late software development stages, performance profiling can be done using trace instrumentation. In our case study we have used trace instrumentation. Traces affect the performance of the system and its usage must be minimized [30,31].

Experience-based assessments are described by Bosch [8]. In this paper, we have described our experience-based assessment method and presented a case study.

Software product family architectures bring several advantages to software development, such as improved software quality and faster derivation of products [7–10]. However, software product family architectures evolve and their evolution has been addressed in various publications. Bosch presented approaches to the adoption of software product family architectures [8,54] and case studies on software product family evolution [55–57].

A recent work has extended the ATAM for the evaluation of software product family architectures [58]. Our work in the software product family domain included software evolution [46] and experiences of software architecture assessments for a software product family [23,27,28].

In general, the scientific literature has explicitly targeted software architecture assessments or software product family architectures but has not addressed the combination of the two research topics. Our experiences of software architecture assessments on software product family architectures aim at increasing the knowledge of the domain.

DOI: 10.1002/smr

8. CONCLUSION

Architecting and maintaining a large software system could be compared to civil engineering, but in actuality the evolution of software design is not mature enough. Problems arise on both the user and the developer sides. On the user side, there are 'unfriendly' user interfaces, software bugs that lead to negative experience in using programs and thus lead to unproductivity. On the developer side, we have delivery dates that are not respected, requirements not satisfied, costs overrun and communication difficulties among team members.

The complexity of the software is an essential property, not an accidental one. Brooks in his famous book [59] points out interesting argumentation supporting this thesis. He wrote a paper 'No silver bullet—essence and accidents of software engineering', and then ten years later wrote 'No silver bullet refired', both of which are included in his book. He argues that 'Not only are there no silver bullets now in view, the very nature of software makes it unlikely that there will be any—no inventions that will do for software productivity, reliability, and simplicity what electronics, transistors, and large-scale integration did for computer hardware.' [59, p. 181]. Ten years later he reached the same conclusion: 'no silver bullet' exists in the software development process.

Is this a pessimistic view? No, it is not. As the author states, skepticism is not pessimistic. There is no easy solution to a complex problem. Several useful technical developments help to create a better software product. High-level languages and the object-oriented paradigm aim to create a higher level of abstraction and to divide the problem into a set of components that interact using specified interfaces and function calls. Software reuse helps to avoid building software already written and tested from scratch; CASE (computer-aided software engineering) environments include compilers, debuggers and editors that together aid in having a better development environment. Model driven methodologies offer a new approach to alleviate the complexity of developing software [60,61]. Extreme programming and agile software development methodologies present an alternative way of developing software, improving its quality through early and continuous software delivery [62–65].

However, one of the most important factors and the most difficult to specify is the human factor because software construction is a creative process. A good programmer can make the difference and great designs come from great designers.

Software architecture assessments constitute an additional aid in the process of software design and evolution and must be included in the process phases as good methodology practice. Assessments performed in the early lifecycle of a project can highlight fundamental architectural flaws; assessments during the lifecycle of the software continue to be essential to verify whether the software system aligns with the requirements. For instance, assessments can be useful when important changes (i.e., due to the evolution) will affect a large part of the architecture.

The assessment of software architectures is a process that is acquiring increasing importance in industrial practice. Researchers in academia and industry are making efforts to establish and create methods of evaluating the strengths and weaknesses of a software product. At the moment one general method does not exist; all of the industries have developed different ways of inspecting the software architecture of the product, even though many similarities exist among these methods.

In this paper we have presented the experiences gathered with three different assessment techniques that we have used successfully in an industrial realm. For each assessment technique we have presented the description of the method, a case study, the results and the lessons learned. The three methods together represent an assessment framework that, if used appropriately, facilitates the software evolution of large software-intensive systems.

DOI: 10.1002/smr

The fact that a large software real-time system part of a software product family architecture was investigated and continuously improved over time demonstrated that continuous assessment brings clear advantages facilitating the software quality and evolution of complex software systems.

Future work will include a formal empirical validation of the assessment techniques used. Continuous assessments also imply continuous improvements and we are continuing to work on the topic and expect to see contributions and experience reports on the subject.

ACKNOWLEDGEMENTS

This work is the result of architecture evaluations carried out by the Nokia Research Center for the Nokia Mobile Phones business unit during a five-year period. In this regard, I would like to thank Andy Turner and Erling Stage for the cooperation and support in our work. I would like to thank Alessandro Maccari who contributed as evaluator to the scenario-based assessment and the experience-based assessment. Thanks to Jan Bosch, Jilles Van Gurp, Jianli Xu, Yaojin Yang, Claudio Riva and all of the anonymous reviewers who have contributed to the improvement of the quality of the paper.

REFERENCES

1. Kruchten PB. The 4 + 1 view model of software architecture. *IEEE Software* 1995; **12**(6):42–50.
2. Bass L, Clements P, Kazman R. *Software Architecture in Practice*. Addison-Wesley: Reading MA, 1998.
3. Perry DE, Wolf AL. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes* 1992; **17**(4):40–52.
4. IEEE Standard. Recommended practice for architectural description of software-intensive systems, 1471-2000, 2000.
5. Glass RL. *Facts and Fallacies of Software Engineering*. Addison-Wesley: Reading MA, 2003.
6. Pigoski TM. *Practical Software Maintenance: Best Practices for Managing your Software Investment*. Wiley: Hoboken NJ, 1996.
7. Jazayeri M, Van Der Linden F, Ran A. *Software Architecture for Product Families*. Addison-Wesley: Reading MA, 2000.
8. Bosch J. *Design and Use of Sofware Architectures*. Addison-Wesley: Reading MA, 2000.
9. Gomaa H. *Designing Software Product Lines with UML*. Addison-Wesley: Reading MA, 2004.
10. Clements P, Northrop L. *Software Product Lines*. Addison-Wesley: Reading MA, 2002.
11. Kang K, Cohen S, Hess J, Novak W, Peterson A. Feature-oriented domain analysis (FODA). *Technical Report CMU/SEI-90-TR-021*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA, 1990.
12. Kang K, Kim S, Lee J, Shin E, Hu M. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering* 1998; **5**:143–168.
13. Griss ML, Favaro J, D'Alessandro M. Integrating feature modeling with RSEB. *Proceedings 5th International Conference on Software Reuse*. IEEE Computer Society Press: Los Alamitos CA, 1998; 76–85.
14. Gomaa H. Reusable software requirements and architectures for family of systems. *Journal of Systems and Software* 1995; **28**(3):189–202.
15. Maccari A, Heie A. Managing infinite variability in mobile terminal software. *Software: Practice and Experience* 2005; **35**(6):513–537.
16. Kazman R, Klein M, Clements P. ATAM: A method for architecture evaluation. *Technical Report CMU/SEI-2000-TR-004*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA, 2000.
17. Kazman R, Abowd G, Bass L, Clements P. Scenario-based analysis of software architecture. *IEEE Software* 1996; **13**(6):47–55.
18. Bengtsson P, Bosch J. Scenario-based software architecture reengineering. *Proceedings 5th International Conference on Software Reuse (ICSR)*. IEEE Computer Society Press: Los Alamitos CA, 1998; 308–317.
19. Rumbaugh J, Jacobson I, Booch G. *The Unified Modeling Language Reference Manual* (2nd edn). Addison-Wesley: Reading MA, 2005.
20. Clements P, Bachmannv F, Bass L, Garlan D, Ivers J, Little R, Nord R, Stafford J. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley: Reading MA, 2002.
21. Medvidovic N, Taylor RN. A classification and comparison framework for software architecture description language. *IEEE Transactions on Software Engineering* 2000; **26**(1):70–93.

DOI: 10.1002/smr

22. ETSI Standard. Digital cellular telecommunication system (phase 2+); General Packet Radio Service (GPRS) service description; stage 2 (GSM 03.60 version 7.4.0 release 1998), 1998.
23. Maccari A. Experiences in assessing product family architecture for evolution. *Proceedings of the 23rd International conference on Software Engineering (ICSE)*. ACM Press: New York NY, 2002; 585–592.
24. The 3rd Generation Partnership Project (3GPP). 3GPP Specifications. http://www.3gpp.org [1 August 2006].
25. Smith CU, Williams LG. *Performance Solutions*. Addison-Wesley: Reading MA, 1995.
26. Del Rosso C. The process of and the lessons learned from performance tuning of a product family software architecture for mobile phones. *Proceedings 8th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press: Los Alamitos CA, 2004; 270–275.
27. Del Rosso C. Dynamic memory management for software product family architectures in embedded real-time systems. *Proceedings 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*. IEEE Computer Society Press: Los Alamitos CA, 2005; 211–212.
28. Del Rosso C. Experiences of performance tuning software product family architectures using a scenario-driven approach. *Proceedings of the 10th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. British Computer Society: Swindon, 2006; 30–39.
29. Del Rosso C. Performance analysis framework for large software intensive systems with a message passing paradigm. *Proceedings of 20th Annual ACM Symposium on Applied Computing*. ACM Press: New York NY, 2005; 885–889.
30. Metz E, Lencevicius R. Efficient instrumentation for performance profiling. *Proceedings of the ICSE Workshop on Dynamic Analysis (WODA)*. ACM Press: New York NY, 2003; 10–12.
31. Metz E, Lencevicius R, Gonzalez TF. Performance data collection using a hybrid approach. *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press: New York NY, 2005; 126–135.
32. Jain R. *Art of Computer System Performance Analysis*. Wiley: Hoboken NJ, 1990.
33. Klein M, Ralya T, Pollak B, Obenza R, Gonzales Harbour M. *A Practitioners Handbook for Real-Time Analysis*. Kluwer Academic: Hingham MA, 1993.
34. Liu CL, Layland JW. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM* 1973; **20**(1):40–61.
35. Fowler M. *Refactoring: Improving the Design of the Existing Code*. Addison-Wesley: Reading MA, 1999.
36. Williams LG, Smith CU. Software performance antipatterns. *Proceedings of the 2nd International Workshop on Software and Performance*. ACM Press: New York NY, 2000; 127–136.
37. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns*. Addison-Wesley: Reading MA, 1995.
38. Noble J, Weir C. *Small Memory Software: Patterns for System with Limited Memory*. Addison-Wesley: Reading MA, 2001.
39. Shaw M, Garlan D. *Software Architecture: Perspective on an Emerging Discipline*. Prentice-Hall: Upper Saddle River NJ, 1996.
40. Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M, Sommerlad P. *Pattern-oriented Software Architecture, Volume 1: A System of Patterns*. Wiley: Hoboken NJ, 1996.
41. Ran A, Lencevicius R. Making sense of runtime architecture for mobile phone software. *Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press: New York NY, 2003; 367–370.
42. Riva C. Reverse architecting: An industrial experience report. *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE2000)*. IEEE Computer Society Press: Los Alamitos CA, 2000; 23–25.
43. Cockburn A. *Writing Effective Use Cases*. Addison-Wesley: Reading MA, 2000.
44. Pyarali I, O'Ryan C, Schmidt DC, Wang N, Gokhale AS, Kachroo V. Using principle patterns to optimize real-time ORBs. *IEEE Concurrency* 2000; **8**(1):16–25.
45. Lehman MM. *Software Evolution*, Marciniak JJ (ed.). Wiley: Hoboken NJ, 1994.
46. Riva C, Del Rosso C. Experiences with software product family evolution. *Proceedings of the 6th International Workshop on Principles of Software Evolution*. ACM Press: New York NY, 2003; 161–169.
47. Clements P, Kazman R, Klein M. *Evaluating Software Architecture*. Addison-Wesley: Reading MA, 2002.
48. IEEE Standard. Standard for software quality metrics methodology, 1061-1992, 1992.
49. Abowd G, Bass L, Clements L, Kazman R, Northrop L, Zaremski A. Recommended best industrial practice for software architecture evaluation. *Technical Report CMU/SEI-96-TR-025*, Software Engineering Institue, Carnegie Mellon University, Pittsburg PA, January 1997. http://www.sei.cmu.edu [1 August 2006].
50. Dobrica L, Niemelä E. A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering* 2002; **28**(7):638–652.
51. Kazman R, Barbacci M, Klein M, Carrière SJ, Woods SG. Experience with performing architecture tradeoff analysis. *Proceedings 21st International Conference on Software Engineering (ICSE'99)*. IEEE Computer Society Press: Los Alamitos CA, 1999; 54–63.

DOI: 10.1002/smr

52. Smith CU, Williams LG. Software performance engineering: A case study including performance comparison with design alternatives. *IEEE Transactions on Software Engineering* 1993; **7**(7):720–741.
53. Williams LG, Smith CU. PASA^{SM}: A method for the performance assessment of software architectures. *Proceedings of the 3rd International Workshop on Software and Performance*. ACM Press: New York NY, 2002; 179–189.
54. Bosch J. Staged adoption of software product families. *Software Process Improvement and Practice* 2005; **10**(2):125–142.
55. Svahnberg M, Bosch J. Evolution in software product lines: Two cases. *Journal of Software Maintenance: Research and Practice* 1999; **11**(6):391–422.
56. Mattsson MM, Bosch J. Stability assessment of evolving industrial object-oriented frameworks. *Journal of Software Maintenance: Research and Practice* 1999; **12**(2):79–102.
57. Bosch J, Bengtsson P. Component evolution in product line architectures. *Proceedings International Workshop on Component-Based Software Engineering*. IEEE Computer Society Press: Los Alamitos CA, 1999.
58. Olumofin F, Misic V. Extending the ATAM architecture evaluation to product line architectures. *Proceedings 5th IEEE/IFIP Working Conference on Software Architecture (WICSA)*. IEEE Computer Society Press: Los Alamitos CA, 2005; 45–56.
59. Brooks PF. *The Mythical Man-Month: Essays on Software Engineering* (20th Anniversary Edition). Addison-Wesley: Reading MA, 1995.
60. Object Management Group (OMG). Model Driven Architecture (MDA). http://www.omg.org/mda/ [1 August 2005].
61. Schmidt DC (ed.). Model driven engineering issue. *Computer* 2006; **39**(2):25–66.
62. Agile Manifesto: Manifesto for agile software development. http://agilemanifesto.org/ [1 August 2006].
63. Beck K, Andres C. *Extreme Programming Explained: Embrace Change*. Addison-Wesley: Reading MA, 2004.
64. Cockburn A. *Agile Software Development*. Addison-Wesley: Reading MA, 2001.
65. Schwaber K, Beedle M. *Agile Software Development with SCRUM*. Prentice-Hall: Upper Saddle River NJ, 2001.

AUTHOR'S BIOGRAPHY

Christian Del Rosso received his MSc degree in Computer Science from the University of Bologna in 2001. Since then he has been a researcher in the Nokia Research Center in the Software and Application Technologies Laboratory. His main research interests are software architecture design and evaluation using qualitative and quantitative methods. In his research work emphasis is given to software product families, software evolution and embedded real-time systems.

DOI: 10.1002/smr