ESPOO 2006 VTT PUBLICATIONS 608

Mari Matinlassi

Quality-driven software architecture model transformation

Towards automation

VTT PUBLICATIONS 608

Quality-driven software architecture model transformation

Towards automation

Mari Matinlassi

Academic Dissertation to be presented, with the assent of the Faculty of Science, University of Oulu, for the public discussion in the Auditorium IT115, Linnanmaa, on September 22nd, 2006, at 12 o'clock noon.

ISBN 951–38–6848–6 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–6849–4 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455–0849 (URL: http://www.vtt.fi/publications/index.jsp)

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

VTT, Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde 020 722 111, faksi 020 722 2320

VTT, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel 020 722 111, fax 020 722 2320

VTT Technical Research Centre of Finland, Kaitoväylä 1, P.O. Box 1100, FI-90571 OULU, Finland
phone internat. +358 20 722 111, fax +358 20 722 2320

Technical editing Maini Manninen

Otamedia Oy, Espoo 2006

Matinlassi, Mari. Quality-driven software architecture model transformation. Towards automation. Espoo 2006. VTT Publications 608. 101 p. + app 95 p.

Keywords software architecture, quality-driven software architecture development, model-driven development, model-driven architecture (MDA), model transformation

Abstract

Model driven software development is about treating models as first class design entities and thereby raising the level of abstraction in software development. A model is a simplified image of a system and, further, model transformation means converting one model to another model of the same system. Transformation is a key to model driven development while automation of transformation is one of the essential goals of model driven architecture (MDA), an initiative to standardize model driven development. Model transformation aims at automating the transition from business models to implementation models. In addition to model refinement, model transformations are used for improving models by restructuring, completing and optimising them.

Quality-driven software architecture model transformation (QAMT) denotes changing an architectural model according to changing or varying quality properties, wherein a quality property is a non-functional interest of one or more system stakeholders. In this dissertation, I examine QAMT automation, i.e. reducing the need for human intervention in QAMT. Therefore, the research question in this dissertation is "how to make automation of QAMT possible". This dissertation provides an answer to the research question by presenting a model to support QAMT automation. The model is derived from the experience gained in four industrial cases and in one laboratory case study. The model is written with Unified Modelling Language 2.0 and includes *activities* to describe the process of transformation and collaborating *actors* that execute the activities.

The goals of the model are (1) to describe transformation as completely as possible, (2) to provide support toward automation, (3) to stay independent of implementation technologies, (4) to be mature and validated and (5) to conform to standards. Transformation is described by presenting a marked model, a mapping and a transformation record, and transformation activities. While the

QAMT model does not support total automation of all the activities, it does reduce the need for human intervention. The QAMT model shows good performance in platform independence and it is validated in five different cases. Finally, the QAMT model promotes understandability by following, e.g., the terminology and specification structures defined in the most important standards in the area.

This research introduces an automation model for quality-driven software architecture model transformation. So far, the research effort on model driven architecture has been focusing on automating vertical transformations such as code generation. The work in this dissertation initiates the automation of horizontal model transformations and suggests future research topics to accumulate the knowledge on the subject and again to derive fresh topics to explore and new ideas to experiment with.

Preface

The work reported in this dissertation was carried out in the Software Architecture group at VTT Technical Research Centre of Finland during 2001–2005. The work was done in the following projects: PLA programme, Minttu1 2001–2002, Minttu2 2002–2003, SUVI 2002, WISE 2001–2003 and FAMILIES 2003–2005.

I am grateful for the financial support provided by Tekes, VTT Electronics, and various Finnish industrial companies. Additional support was received from Nokia Foundation, which awarded a scholarship for the work in 2004.

I wish to thank Professor Eila Niemelä for supervising my work and teaching me many important things needed in the career of a researcher. I also wish to thank the co-authors of the publications included in this dissertation and special thanks go to the personnel involved in the projects mentioned above for creating a friendly and inspiring atmosphere in which to work.

The manuscript of this dissertation was reviewed by Professor Jan Bosch and Professor Philippe Kruchten. I am grateful for their expert comments which led to remarkable improvements of the thesis. I also wish to express my sincere gratitude to Professor Eila Niemelä for her valuable comments and constructive criticism on the earlier versions of the manuscript.

Last but not least, I'd like to express my warm and deep gratitude to my family Mika, Sampo, Joona, Otso and Sini for their patience during these years and special thanks for providing a colourful counterbalance to – even so interesting – research work. Thanks to children I have learned one of the most important Jedi wisdom: "Try not. Do. Or do not. There is not try."

Ii, June 2006

Mari Matinlassi

Contents

Appendices III and IV of this publication are not included in the PDF version. Please order the printed version to get the complete publication (http://www.vtt.fi/publications/index.jsp)

List of original publications

I Matinlassi, M. 2004. Comparison of software product line architecture design methods: COPA, FAST, FORM, KobrA and QADA. Proceedings of the 29th international conference on software engineering, ICSE2004. Edinburgh, Scotland, U.K., 23–28 May 2004. Los Alamitos, California: IEEE. Pp. 127–136. ISBN 0-7695-2163-0.

II Matinlassi, M. & Niemelä, E. 2002. Quality-driven architecture design method. Proceedings of the 15th international conference of software & systems engineering and their applications, ICSSEA 2002. Centre pour la Maîtrise des Systèmes et du Logiciel Conservatoire National des Arts et Mètiers, Paris, France, 3–5 December 2002. Paris, France: CMSL. Vol. 3. Session 11. ISSN 1637-5033.

III Purhonen, A., Niemelä, E. & Matinlassi, M. 2004. Viewpoints of DSP software and service architectures. Journal of systems and software, Vol. 69, No. 1–2, pp. 57–73. ISSN 0164-1212.

IV Lago, P. & Matinlassi, M. 2002. The WISE approach to architect wireless services. In: Oivo, M. & Komi-Sirviö, S. (eds.). Proceedings of the 4th international conference in product focused software process improvement, PROFES'02. Rovaniemi, Finland, 9–11 December 2002. Berlin, Heidelberg, Germany: Springer-Verlag. Pp. 367–382. (Lecture Notes in Computer Science 2559.) ISBN 3-540-00234-0.

V Matinlassi, M. & Niemelä, E. 2003. The impact of maintainability on component-based software systems. In: Chroust, G. & Hofer, C. (eds.). Proceedings of the 29th EUROMICRO conference, New waves in system architecture. Belek-Antalya, Turkey, 1–6 September 2003. Los Alamitos, California: IEEE Computer Society. Pp. 25–32. ISBN 0-7695-1996-2.

VI Niemelä, E., Matinlassi, M. & Lago, P. 2003. Architecture-centric approach to wireless service engineering. Annual Review of Communications, Vol. 56, pp. 875–889. ISBN 1-931695-22-9.

VII Tikkala, A. & Matinlassi, M. 2002. Platform services for wireless multimedia applications: case studies. In: Ojala, T. & Ollila, M. (eds.). Proceedings of the 1st International Conference on Mobile and Ubiquitous Multimedia, MUM 2002. Oulu, Finland, 11–13 December 2002. Oulu: Oulu University Press. Pp. 76–81. ISBN 951-42-6909-8.

The author of this dissertation is the principal author in Papers I, II and V. The author's effort for the rest of the papers has also been essential by providing design method expertise as well as being a co-author.

Abbreviations

ADL	Architecture Description Language
API	Application Programming Interface
CASE	Computer Aided Software Engineering
CIM	Computation Independent Model
CORBA	Common Object Request Broker Architecture
DPS	Dual Protection Style
DS	Dissertation Summary
DSP	Digital Signal Processing
DTD	Document Type Definition
FAMILIES	Fact-based Maturity through Institutionalisation Lessons-learned and Involved Exploration of System family engineering
ID	Identification
IEEE	Institute of Electrical and Electronics Engineers
ISO/IEC	International Organization for Standardization/ International Engineering Consortium
J2EE	Java 2 Platform Enterprise Edition
MDA	Model Driven Architecture
MDD	Model Driven Development

MDSD	Model Driven Software Development
MOF	Meta Object Facility
MQ	Message Queuing
MSMQ	Microsoft Message Queuing
OCL	Object Constraint Language
OMG	Object Management Group
PC	Personal Computer
PDA	Personal Digital Assistant
PF	Product Family
PFA	Product Family Architecture
PIM	Platform Independent Model
PSM	Platform Specific Model
ROOM	Real-time Object Oriented Modeling
SQL	Structured Query Language
QADA[1]	Quality-driven Architecture Design and quality Analysis
QAMT	Quality-driven Architecture Model Transformation
QVT	Query/Views/Transformations

[1] http://virtual.vtt.fi/qada

SWA	Software Architecture
TDL	Transformation Description Language
UML	Unified Modelling Language
UML2	Unified Modelling Language, version 2.0
XML	eXtensible Markup Language

1. Introduction

Software intensive products have won popularity in everyday life today. An increasing need for faster, cheaper and even more versatile software intensive products sets a real challenge for the software industry. The software industry is constantly looking for ways to improve the cost-effectiveness of software development and the quality of software products.

The dissertation summary (DS) presents a model for quality-driven software architecture model transformation (QAMT). QAMT denotes changing an architectural model according to changing or varying quality properties, wherein a quality property is a non-functional interest of one or more system stakeholders. The aim of developing the QAMT model is to promote automation of transformation and thereby making changing software architecture easier. Reducing the need for human interaction in transforming an architectural model improves the cost-effectiveness and quality of software products.

In this section, I first provide definitions for the most important terms used throughout the dissertation summary as an *introduction to the topic*. Then, QAMT *motivation* is presented to illuminate the need for QAMT, to locate the research gap and to reason why we shall aspire after automation, in particular. The *research problem* and the *limitations* of the study are discussed next with a short overview on the research *results*. Thereafter, the *research approach* is presented including a description on how the research was conducted, how each of the original publications contribute to the research and what kind of cases were used to derive and validate the research results. The *outline of the dissertation* provides an overview of the dissertation summary.

1.1 Introduction to the topic

Software architecture is an essential part of software intensive products. Software architecture is the structure or structures of the system including components, their relationships to each other and to the environment (Bass et al. 1998). Software architecture also includes the principles guiding its design and evolution (IEEE-1471 2000). In the same way as construction architecture strongly influences the properties of a building, software architecture has a strong influence on the life cycle of a software system. Therefore, *software architecture development* may be considered as one of the most important issues in developing high quality software products.

Quality-driven software architecture development emphasizes the importance of qualities, wherein qualities refer to the non-functional properties of software products. The approach relies on gathering, categorizing and documenting quality properties as at least equally important requirements as functional requirements and constraints, and utilizing the gained knowledge in architectural design. The quality-driven design is further complemented with an *architectural analysis*.

Architectural analysis is about testing the *architecture model* produced in the design, i.e. verifying whether the architecture meets the quality requirements set in the very beginning.

A Model is a simplified image of a software system. A model is written in the language of its unique *metamodel,* and a model cannot be understood and has no meaning when separated from its *metamodel*. *Metamodel* provides the language or legend for understanding a *model*.

Architecture model is a description of software structures, represented with one or more architectural *views*, wherein architectural *view* represents a whole software system from the perspective of a related set of concerns (IEEE-1471 2000). *Views* may also abstract the details away from the system and therefore a view may include more than one abstraction level. A view may consist of one or more architectural diagrams. Each architectural diagram is developed using the methods established by its associated architectural *viewpoint*. *A viewpoint* establishes the conventions by which a *view* is created, depicted and analyzed.

Model-driven software development (MDD) focuses on providing models – rather than programs – as primary software products. Modelling provides a less risky, more cost-efficient and easier to understand view to a complex problem and its solutions than implementing a genuine target (Selic 2003). *Model-driven architecture*, MDA (OMG 2003a), is an OMG (Object Management Group) initiative designed to provide a standardization framework for MDD. MDA expresses model abstraction as *platform independence*. In the context of software development, platform denotes information-formatting technologies, 3rd and 4th generation languages, distributed component middleware and messaging middleware (Frankel 2003).

Model transformation is the process of converting one model to another model of the same system (OMG 2003a). The definition of *model transformation* above does not define whether the model is converted to another model manually, automatically or semi-automatically. In this dissertation the transformation concept of the MDA approach is applied, wherein transformation especially aims at automation. Transformations are defined with *rules*. A *rule* is responsible for transforming a particular selection of the source model to corresponding target model elements (Gardner et al. 2003). The inputs for transformation are a *marked* model (source model) and a *mapping* whereas the transformation outputs are a target model and a transformation *record*.

A *mark* represents a concept in the target model, and is applied to an element of the source model (OMG 2003a). Mark only *indicates* how an element is to be transformed. The actual transformation specification is provided in *mapping.* A mapping is specified using some language to describe a transformation of one model to another. The description may be in natural language, an algorithm in an action language, or in a model mapping language (OMG 2003a).

The *record* of transformation includes a map from the element of the source model to the corresponding elements of the target model, and shows which parts of the *mapping* were used for each part of the transformation (OMG 2003a). A record consists of *traces.* A *trace* (OMG 2005a) records a link between a group of objects from the source models and a group of objects in the target models.

Horizontal model transformation represents a special type of model transformation. *Horizontal transformations* do not affect model abstraction level (Cristoph 2004) and are used to restructure, complete, or optimise a software model in order to improve its internal structure and/or quality (Ramljak et al. 2003). *Vertical transformations, by contrast,* do affect the model abstraction level and they are used to refine or to abstract a model during forward or reverse engineering.

Architecture model transformation is a horizontal model transformation at the abstraction level of platform independence, transforming an architecture model to another model at the same level of abstraction. In addition, architecture model transformation aims at automation. In the case that the trigger for architecture model transformation arises from the changing or varying of quality requirements, the transformation is referred to as *quality-driven software architecture model transformation,* hereafter briefly QAMT.

The changing or varying of quality requirements, i.e. quality variability, occurs especially in the context of *product families*. A *product family* or a *product line* generally means a group of partly similar software products developed with specialized methods. *Product family* and *product line* are often used as synonyms in spite of the fact that there is a clear distinction between these two concepts.

Software product line refers to engineering techniques for creating a collection of similar software systems from a shared set of software assets using a common means of production (Krueger 2004). *Software product line* is thus a process-oriented term emphasizing the inputs and outputs of the development process and also the decisions made and artefacts created on products in the various phases and activities of the development process.

Software product family, for its part, refers to a group of products sharing a common, managed set of features satisfying the specific needs of a given market. *Software product family* further consists of a *product family architecture* and a set of reusable components designed for incorporation into the product family architecture (Bass et al. 1998; Bosch 2000). Software product family *members* are instances of a product family. Software product family can thus be regarded as a product-oriented term emphasizing the business and organizational aspects of a product family in addition to its process and architecture related features (van der Linden et al. 2004).

Product family architecture (PFA) is an adaptable form of architecture, which is applied to the product members of a product family and from which the software architecture of each product member can be derived. PFA is typically software architecture involving a set of reusable components shared by a family of products (Bass et al. 1998).

1.2 Motivation

Referring to the American Programmer journal, Abrahamsson (2002) points out that "Everyone knows the best way to improve software productivity and quality is to focus on people."

I must agree with the statement above. However, focusing on people is not enough. Even the best people need methods, techniques and tools to complete their work. Methods (Kronlöf 1993) guide people to conduct a process with steps and examples. Methods further define what language to use and also include techniques for e.g. describing software with architectural viewpoints. Tools complement and automate the techniques and methods. Simple tools only help people in their work whereas advanced tools do all or most of the work on behalf of people.

Any manually conducted work (with simple tools) requires more time and effort than semi-automated or automated work (with advanced tools). Similarly, manually conducted QAMT requires more time, effort and money than automated QAMT, while QAMT can definitely be regarded as an unavoidable part of the software product life cycle. From this point of view, the need for automated QAMT is quite obvious. Automated quality-driven software architecture transformation will make the architecture development process faster and easier for the architect and therefore cheaper for the company. With automated transformation, the software architect may also easily try out the feasibility of various architectural options. The ease of automated development is due to the possible simulation capabilities of the architectural model that will retire the "architect, design, implement, test and start it all over" development process and replace it with a more advanced and automated "model, simulate, generate" development process. The ease of testing more architectural options than earlier will also increase the quality of software architecture. The more comfortable it is to try out various options the more likely it is for a software architect to test more options. The more various options are studied the more probable it is that a better one than the current will be found.

Quality-driven architecture model transformation provides its benefits especially in the context of product families. The products of a product family may involve various customer groups desiring different functions and qualities for a product.

For instance, one customer may value reliability while considering response time not so important, whereas another customer may be requesting a short real-time response time. As another example, different environments of use may affect the variability in the quality aspects of a product family. In other words, a product may involve varying hardware, portable and fixed-point, while its software functionality remains unchanged. Despite common functionality the products may also have varying security requirements. In addition, portable devices (e.g. Personal Digital Assistants, PDAs) are evolving more quickly than fixed-point devices (e.g. Personal Computers, PCs). This rapid evolution requires a PDA software product to be as platform independent as possible or at least to provide a high degree of portability. Automated QAMT would enable easier optimisation and modification of individual product qualities.

So far, the research community has been focusing on, for example, applying MDA in various application domains, defining transformation languages for vertical transformations and UML (Unified Modelling Language) profiles (see e.g. (Aßmann 2004)). The *quality-driven architecture model transformation* concept has not, however, received any greater attention so far. Christoph (2004) presents a rule-based transformation framework, which facilitates horizontal or vertical transformations among UML 1.1 models, focusing on class-diagrams. This approach does not, however, consider the effects of quality requirements on horizontal transformation, and, furthermore, the framework applies an obsolete version of modelling language, which does not support architecture modelling. Bosch & Molin (1999) present a cyclic transformation process for improving the quality characteristics of software architectures. Here the basic idea of architectural transformation is that functional requirements have already been fulfilled, while quality characteristics are determined through an evaluation of the architecture and possible unmet quality requirements are achieved through restructuring i.e. transforming the software architecture. This approach does not directly deal with model-driven software development and it only considers manual architecture transformation.

Grunske (2003) presents an approach towards automation of architecture transformation. The approach utilizes a hypergraphs theory for automating UML-RT model transformation. Here, the automation concentrates only on transforming the graphical representation of architecture and checking the behavioural equivalence between source and target with a proof algorithm.

Architecture evaluation (i.e. identifying source model and target model) is mentioned but not dealt with in any detail.

One of the fundamental ideas of model-driven development is that more modellers than developers will be needed in the future. The more modellers, the more automated modelling tools are required. Furthermore, no technique, including the architecture transformation technique, is effective without appropriate tool support. Tools providing support or guidance to the modeller and enabling automation of, for instance, traceability and transformation are, as stated in (Steenbergen et al. 2004), the key for an industrial adoption of MDA. Thus, the automation of MDA should be one of the main research issues at the moment.

1.3 Problem, limitations and results

Converting architectural models to other models is likely to be a common routine for developers in the software industry. Models are constantly manually converted during the development process due to various reasons. Model conversion may be, for example, due to changes in the business or in functional or quality requirements. Thus, as QAMT already is possible and conducted constantly in the software product life cycle, although manually, this research does not consider the question of *how to make quality-driven software architecture model transformation possible.* The specific problem to be studied in the dissertation is:

How to make automation of quality-driven software architecture model transformation (QAMT) possible?

Semi-automated or even automated quality-driven software architecture transformations are a part of an ideal state of the model-driven architecture approach. Currently, the maturity of the different specifications included in the MDA standardization framework varies. The development of standards for automated transformations has to be started and indeed has been started at the grassroots level, nearer to platform dependence than platform independence. As the semantic expressiveness of the unified modelling language increases (Kobryn 2004), the future of developing standards for automated transformations will be focusing on the higher levels of platform independence. Over the past few years, the level of abstraction for software practitioners in software descriptions has been increasing (Brown 2004). Within MDA, a Computation Independent Model (CIM) has already been defined, focusing on the environment of the system and the requirements for the system (OMG 2003a). If we assume that the development trend of model abstraction does not change much in the not so far future, automated transformations may be realized at the architecture level or even at the level of product requirements. However, in the current circumstances the scope of this dissertation is not "how to *make* the automation", because the problem then would cover several implementation technologies and therefore be far too large to be studied in this dissertation summary.

There is no fixed way to describe a software architecture model among software professionals. Automatic conversion of models is a complicated task and the variance in model descriptions certainly does not make it easier. Therefore, this dissertation is restricted to architectural models described with the QADA® methodology (Quality-driven Architecture Design and quality Analysis). The ideology of the QADA methodology is introduced in Section 2.1 and also discussed in Papers I–VII.

Finally, the intention of this dissertation is neither to develop new nor to improve existing patterns for quality-driven architecture modelling but rather to assume that the number and scope of the currently available patterns is extensive enough to be utilized in this research. Further, the intention is not to develop any mapping language, i.e. transformation description language (TDL), either.

Quality-driven architecture model transformation may be easily confused with refactoring, because both approaches, to put it in general terms, change software structure without changing functionality. However, the abstraction level of refactoring is closer to implementation than architectural models. Refactoring directly addresses code, whereas architecture model transformation deals with models. In other words, quality-driven architecture model transformation may be performed early in the architecting phase whereas refactoring changes an existing and already implemented software. In addition, in a special case an architectural transformation may change functionality through indirect functional variability. In other words, the variability of quality requirements causes indirect variability in functionality, e.g. reliability requirement changes, and a new fault treatment mechanism is added to the architecture. Thus, the question in architecture model transformation is not only about changing structure, but also changing and improving quality.

After discussing the research question and its limitations above, I provide an overview of the results of dissertation research. This research gives an answer to the research question by providing a model for QAMT automation. The goals of the model are:

- Provide as complete as possible a transformation specification, while preserving platform independence.
- Promote automation of validated transformation process.
- Retain understandability of the model by utilizing current state-of-practice and standards in specification structure, specification language and terminology.

The QAMT model includes *activities* for describing the transformation process and transformation *actors* for representing the collaborating objects that execute the transformation.

1.4 Research approach

The work done in this dissertation is a part of a long-term research started in 2000, namely the development of the QADA® methodology. The development is done in a sequence of various types of research projects involving several researches, each project and researcher focusing on certain part(s) of the methodology. The research approach of the whole concept is to create, validate and improve parts of the methodology as methods, techniques and realizations, to evaluate the parts and therefore to iteratively elaborate the methodology. Methodology parts are individual methods, wherein a method (Kronlöf 1993) denotes (1) an underlying model, (2) a language, (3) defined steps and ordering of these steps and (4) guidance for applying the method complemented with (5) tool support. My research efforts have concentrated on the development of the quality-driven architecture *design* method and its constructs as an integrated part of the QADA methodology.

This dissertation research applies the constructive research approach (Järvinen 2004). My research covers three main steps, which iteratively improve and extend the method as follows. (1) **Collect data** by studying the state-of-the-art in the specific field and/or make empirical observations in cases, (2) **develop method constructs,** i.e. analyse collected data and produce parts of the method such as steps, model, notation, tool support and guidance and (3) **test method constructs** with multiple cases (van Aken 2004), self-evaluations and comparisons (Järvinen 2004). Testing the method with a case also serves for data collection (step 1) for the next method iteration.

The first method constructs were originally introduced in (Matinlassi et al. 2002), after which they have been tested in cases (Papers IV–VII) and comparisons (Papers I & III), iteratively improved (Paper II) and extended (Paper III) to finally produce the QAMT model presented in this dissertation summary. Table 1 summarizes the cases that were used for QAMT data collection and design method testing. The cases are identified (IDs) as C1 (case 1) to C9 (case 9).

Table 1. Summary of cases used in the research.

Domain	**ID**	**Type**	**Description**
Middleware services	C1	Laboratory, pf (product family)	Distribution service management platform
	C2	Industrial	Multimedia streaming service
	C3	Industrial	Instant messaging and presence service
Wireless services	C4	Research pilot, ref. architecture	Multiplayer game
Terminal software	C5	Industrial pf	Terminal software for a fare collection system used in public transportation
Control systems	C6	Industrial pf, third party case	Prototyping framework for a family of multifunctionals
Telecommuni-cation	C7	Industrial, pf, third party case	Analyser and simulation software product families for telecommunication networks
	C8	Industrial, pf	Base transceiver station family
Measurement	C9	Industrial pf	Measurement system family

The first one of the cases, C1 (Matinlassi et al. 2002, Paper II) was the starting point for the development of the design method. C1 was also used as a starting point for the development of automated QAMT (Merilinna 2005). The aim of the further cases in the domain of middleware services, namely C2 and C3 (Paper VII), was to test the method in industry, especially in small development teams. Large development teams become familiar in test case C4 (Papers IV & VI) wherein pilot wireless services (shortly pilots) were developed in a European, multinational and multi-site development team. The terminal software product family in C5 (Matinlassi 2004, Paper V) tested the design method in a middle sized, local development team. In addition to design, quality analysis was also applied in C5. Next, it was time for third party cases C6 and C7. In case C6 (Vrijnsen et al. 2003), in the domain of control systems, a prototyping framework was developed for a family of multifunctionals. Third party representatives studied and applied the design method almost independently with minor guidance from a colleague of the author. Case C7 was also a third party case. In this case, the company representatives studied and applied the architecture design method providing conceptual and concrete architecture designs. The author reviewed both architecture documents, and after feedback meetings the company carried on with architecture development, design and implementation independently. After applying the design method in cases C6 and C7, the representatives of the case companies answered an experience questionnaire in the Web[2] with 43 questions. The questionnaire study provided feed back for design method development. Case C8 (Matinlassi et al. 2004) adapted the method viewpoints to the base transceiver station family. Case C9 applied the design method for a measurement system family. Cases C1 - C5 are selected as real cases for collecting data and validating QAMT. Cases C6 - C9 are not valid for the purposes of this thesis because they are either third party cases (control systems, telecommunication) or not published (telecommunication, measurement systems). Table 2 shows the contribution of each case to QAMT.

[2] http://cgi.vtt.fi/html/kyselyt/qada/

Table 2. Contribution of cases C1 - C5 to the QAMT model.

Case	Empirical observation(s) = Contribution
C1	Trial to execute transformation in a laboratory case. Observed transformation triggers: quality variability in time, i.e. quality requirements evolution. Observed how QAMT was performed manually. The first implementation trial of the QAMT automation model.
C2, C3	Trial to transform architecture from case C2 to C3. Observed transformation triggers: functional variability in space, no quality variability (static quality attributes: modifiability, integrability and portability), architectural transformation not relevant.
C4	Trial to develop a pilot and to transform the architecture twice. Observed transformation triggers: quality requirements evolution in pilot iterations (first transformation: real-time performance, second transformation: modifiability)
C5	Trial to evaluate architecture, suggest appropriate transformations and estimate the effects of transformations. Observed transformation triggers: quality variability in space (e.g. security requirements are different in differing environments) and quality variability in time (e.g. extensibility for future functions).

Next, it is described – in more detail – how the dissertation research was conducted (Figure 1). The illustration presents how empirical data was collected, analysed, reported and the results tested in the next case, which again also acted as a source for empirical data of the next method iteration (Figure 1 a). The outermost circle represents the cases. The inner three circles represent the research topics, which are built on top of each other as can be seen in Figure 1b. In manual quality-driven software architecture model *transformation,* the software architect needs information on the *model* and especially on the *architecture quality* in the model. In other words, information is needed on topics such as the abstraction level of modelling, architectural viewpoints, modelling language, applied diagrams, as well as on the quality requirements, quality variability and quality representation of the model. All the papers I–VII are related to the topics stated above. In particular, the dissertation summary utilizes empirical experience gained in testing the previous versions and parts of the design method in multiple cases and also interprets the old case results in a new way and thus produces an inductively derived model. The QAMT model partly describes the underlying model of quality-driven design thus contributing to the QADA methodology as one method construct. The model itself is tested in the dissertation summary through assessing it against evaluation criteria.

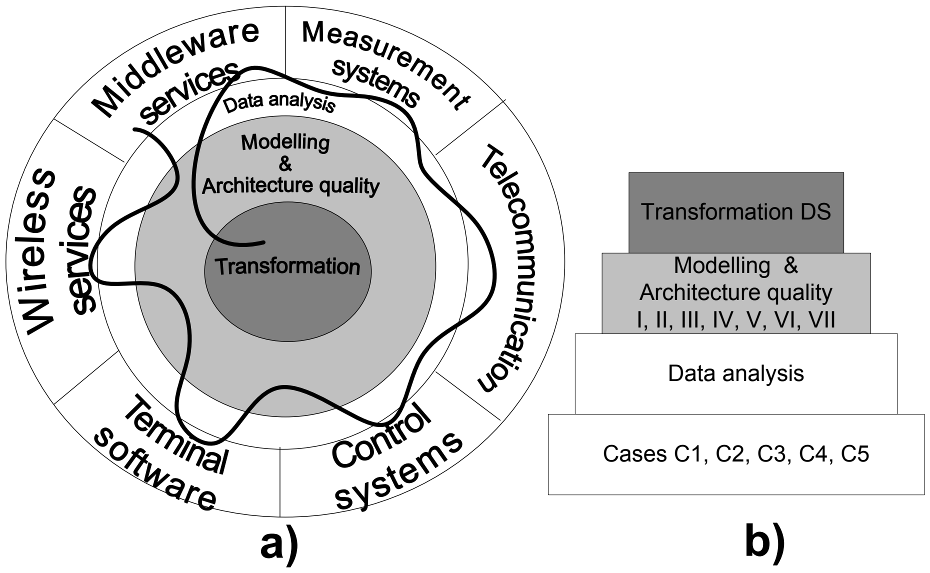


Figure 1. Dissertation research path.

Table 3 classifies the papers I–VII, other publications related to dissertation research and the dissertation summary according to the research topics and research steps mentioned above.

Table 3. Summary of publications related to dissertation research.

Step \ Topic		Architecture quality	Modelling	Transformation
Study state of the art		V	I, III, (Matinlassi 2002)	DS, (Merilinna & Matinlassi 2004)
Develop method		II, V, VI, (Matinlassi & Niemelä 2002)	II, III, IV, VI (Matinlassi et al. 2002)	DS, (Matinlassi 2005)
Test	**Case**	V, VII (Matinlassi 2004)	IV, VI, (Matinlassi et al. 2004; Niemelä et al. 2004, Vrijnsen et al. 2003)	DS, (Merilinna 2005)
	Evalu-ation	-	I, III (Matinlassi & Kalaoja 2002)	DS

Table 4 further clarifies the contribution of Papers I to VII for the dissertation research. The method constructs in Table 4 are based on the definitions by Kronlöf (1993), America et al. (2000) and March & Smith (1995). The short names and their descriptions for constructs are as follows:

1) Model = a description of "how things are". Forms a vocabulary of the method and constitutes the concepts for understanding the method. The representation of the model is not constrained any way.
2) Language = how to describe software architecture. In addition to bare language, this construct describes the viewpoints, diagrams etc. needed to describe software architecture.
3) Steps and their ordering = how to design software architecture. Includes overall design phases and an easy to follow 1-2-3 step for completing the phases.
4) Guidance = Illustrative example(s) on using the constructs above.
5) Tool support = A CASE (Computer Aided Software Engineering) tool, a set of CASE tools and/or tool extensions for supporting the models, steps and language above.

Table 4. Contributions of Papers I–VII and DS for dissertation research.

Research step		Ref	Contribution
Study state of the art		I	Evaluate existing SWA (software architecture) development methods
		III	Evaluate existing SWA description techniques
		V	Study state-of-the-art of quality attributes
		DS	Study state-of-the-art of quality-driven architecture model transformation
Develop method construct	Steps, language	II	Develop a method for designing and modelling software architecture for products and product families
	Language	III	Improve the method with a fourth SWA description viewpoint
	Steps, language, guidance	IV	Adapt the method to modelling especially wireless services
	Model	V	Constitute the concepts for quality-driven architecture design: impact of maintainability
	Model, language	VI	Illustrate two separate abstraction levels and four viewpoints Constitute the concepts for quality-driven architecture design: quality stack
	Model	DS	Elaborate the method with QAMT automation model
Test	Case	II	Introduce C1 and report future research based on case experiences
		IV, VI	Test quality-driven design in large development team in the domain of wireless services, C4
		V	Test quality-driven design and quality evaluation in a product family with middle size development team in the domain of terminal software, C5
		VII	Test quality-driven design in small development teams in the domain of middleware services, C2, C3
		DS	Interpret old case results in a new way to derive QAMT automation model, C1–C5
	Evaluate	I	Compare the design method with other similar methods
		III	Compare QADA viewpoints with DSP (Digital Signal Processing) viewpoints
		DS	Evaluate the QAMT model against criteria

1.5 Outline of the dissertation

Section 2 provides a reference framework and the state-of-the-art on quality-driven software architecture model transformation. The concept is approached by first introducing quality-driven software architecture development and then by concentrating on architecture modelling. Further, architecture model transformation is discussed by presenting a small taxonomy of model transformation and introducing the most important approaches to transforming an architectural model.

Section 3 describes the QAMT model by first introducing the main concepts related to it, i.e. activities and actors, and illustrating how the actors collaborate to execute transformation. After that, an automation model is presented for each of the main activities as an UML activity diagram. Finally, the section concentrates on the two most important automation actors and defines (1) what information shall an actor contain and why, (2) how information is captured in a uniform way and (3) how information is obtained.

Section 4 introduces the evaluation criteria and an assessment of the model against the defined criteria. The evaluation criteria, as derived from the needs of the QAMT model stakeholders, are the following: completeness of the transformation specification, platform independence of transformation actors, automation level of activities, maturity of activities and conformance to standards. The results are presented in the assessment section and in the evaluation summary. The evaluation summary also discusses the consistency of the evaluation results.

Section 5 presents the conclusion of the dissertation summarizing its results, discussing the limitations of the results and drawing out some points for future research.

Section 6 is an introduction to the original papers included in this dissertation. The main considerations of the papers are summarized and papers are categorized into state-of-the-art, method development and case papers.

Papers I to VII are presented in appendices.

2. Quality-driven software architecture model transformation

This section discusses the state-of-the-art of quality-driven software architecture *development* and then especially focuses on software architecture *modelling* and architecture model *transformations*. The main concepts of the approach are first introduced, concentrating on quality properties, quality property variability in software architectures and how quality is represented in models. The notions of model, architectural model, model-driven development and especially MDA are discussed in the SWA modelling section. The transformation section includes two main topics: the taxonomy of model transformations and an introduction to the most important architectural model transformation approaches.

2.1 Quality-driven software architecture development

The basic principle of quality-driven architecture development is to emphasize the importance of quality properties. This is realised by gathering, categorizing and documenting quality properties as at least equally important requirements as functional requirements and constraints (Paper II), and utilizing the gained knowledge in architecture design. The quality-driven design is further complemented by an architectural analysis evaluating the models produced. In practice, the quality-driven approach in software development is realized in several ways (Paper III, Paper VI):

- *Emphasizing quality properties in eliciting requirements.* Eliciting and mapping quality requirements through the design, from the requirements to the architectural model.
- *Modelling architecture with emphasis on quality properties.* Describing software architecture with viewpoints dedicated to certain quality attributes and representing quality in viewpoints with styles, patterns and quality profiles.
- *Validating that quality properties are realised in models.* Utilizing architectural viewpoint descriptions in the evaluation, the designed architecture is validated against the quality requirements set in the beginning (Dobrica & Niemelä 2000; Dobrica & Niemelä 2002).

The next sections focus on those aspects of quality-driven software architecture development that are closely related to architecture model transformation: quality properties and their variability as well as the quality representation in the model.

2.1.1 Quality properties

Functional requirements specify functions that the developed software must be capable of performing. *Non-functional requirements* in software development describe *how* functional requirements shall be realized in the software product(s) (Chung et al. 2000). In other words, non-functional requirements cover the full spectrum of software development and production including non-functional requirements from various perspectives such as business, development and user, thus including such non-functional requirements as development cost, project stability, maturity and learnability.

The ISO/IEC (International Organization for Standardization/ International Engineering Consortium) Quality model 9126-1 (2001) divides product quality attributes into two categories: quality attributes of an intermediate product (i.e. while the product is in production) and those of a completed product (i.e. while the product is ready and in use). The quality attributes for intermediate products are: functionality, reliability, usability, efficiency, maintainability and portability. The quality attributes for a product in use are: effectiveness, productivity, safety and satisfaction.

Intermediate product quality is further divided into *internal* and *external* quality. Different metrics are used to evaluate internal and external quality although the names of quality attributes remain the same. How I see it, ISO 9126-1 (2001) is a process-oriented standard looking at things through the software process. According to this standard, *internal* refers to process phases where you are able to access the “fundamental design” (i.e. code) whereas *external* represents the process phases where you are not any more able to access the “fundamental design” (e.g. in testing phase), while small improvements are still possible.

The software architecture community talks about architectural quality attributes (Bass et al. 1998), which are the non-functional requirements especially related to software architecture. The IEEE (Institute of Electrical and Electronics

Engineers) standard 1471 (2000) completes the definition of quality attribute (although calling it ‘architectural concern’) as follows “..those interests which pertain to the system’s development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders.” Examples of these architectural concerns include performance, reliability, security, distribution, and evolvability.

Quality attributes are often classified into two main categories (Bass et al. 1998; Dueñas et al. 1998): development and evolution time qualities, and execution qualities. Execution qualities, e.g. reliability and performance, are discernible at run-time, whereas development and evolution qualities – such as extensibility or integrability – are considered in the architecture development (Paper V). Compared to the quality attribute categorization in ISO 9126-1, this categorization is independent of process phase and is based on the twofold nature of software architecture, i.e. static – dynamic (Bratthall & Runeson 1999).

Due to the variance in terminology and orientation of software quality, Figure 2 illustrates the approach used in the summary of this dissertation. The term quality property is here used to refer to quality attribute, quality requirement or the combination of an attribute and corresponding requirements.

The two most important sources for quality attributes and requirements are (1) stakeholder needs and (2) application domain (Paper VI; Niemelä and Matinlassi 2005; Al-Naeem et al. 2005). Quality attributes and requirements are defined by identifying the interested stakeholders and their targets concerning the product family. In other words, quality property is *a non-functional interest of one or more stakeholder(s)*. Product quality attributes are also derived from the domain quality attributes. That is, certain qualities are typical of the domain while certain attributes are considered less important. An application domain is a specific group of software systems – such as medical systems, measurement systems, distributed systems and information systems – or a specific group of software components in a system e.g. graphical user interface domain. Because of the large diversity in quality attributes and their definitions, the quality attributes used in this work are described in Table 5.

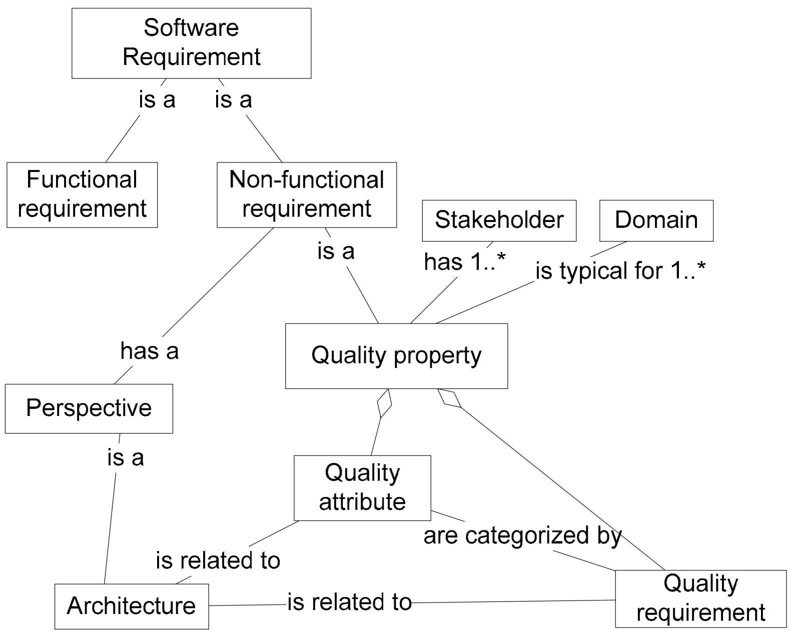


Figure 2. Small quality terminology.

Paper V defines the relationships between the evolution quality attributes listed in Table 5 as follows. Maintainability is the ease with which a software system or component can be modified or adapted to a changed environment. Therefore, the definition of *maintainability* is very close to *modifiability*. Modifications include adding new components (requires *extensibility* and *integrability*), porting to different computing system (requires *portability* and *flexibility*) and deleting unwanted functions. No matter what kind of changes the system is subjected to, it must be tested after the changes have been made (requires *testability*). Furthermore, any software with a long life-cycle requires *reusability* (e.g. use of standards, component-based development and up-to-date component documentation) from the system components. To conclude, all the attributes in Table 5 are more or less related to maintainability.

Table 5. Evolution quality attributes.

Attribute	**Description**
Maintainability	The ease with which a software system or component can be modified or adapted to a changed environment.
Modifiability	The ability to make changes quickly and cost-effectively.
Extensibility	The system's ability to acquire new components.
Integrability	The ability to make the separately developed components of the system work correctly together.
Portability	The ability of the system to run under different computing systems: hardware, software or combination of the two.
Flexibility	The ease with which a system or component can be modified for use in applications or in an environment other than those for which it was specifically designed.
Testability	The ease with which software can be made to demonstrate its faults.
Reusability	The ability of system structure or some of its components to be reused in future applications.

2.1.2 Variability in quality properties

There seems to be disagreement within the research community on whether variability is a quality requirement (Chung et al. 2000; Purhonen 2002) or not (Salicki & Farcet 2001). Referring to Salicki and Farcet (2001), *variability is not a quality factor as such*, but it provides a mechanism for managing the anticipated changes in software structure(s) during the evolution of systems. Thus, how I see it, variability is used as a mechanism to improve other quality properties such as maintainability and extensibility – especially in product families. Very few publications concern *variability* especially *in quality properties*. For example, the future research plan of Andersson and Bosch (2005) is as follows: "We also plan to study how variability management can be improved for non-functional requirements and carry out a more in-depth study of the dynamism aspect and how this is managed in the architectural design process." Therefore, I here discuss and provide a short summary on those properties of functional variation that are applicable to quality variation. Quality variation is here not regarded as an unwanted diversity of quality but as an *intentional* variation in stakeholders' non-functional interests.

In order to derive varying products from a PFA, software product family architecture has to support variability of functionality and quality in space and time (Bosch 2000). Variability *in space* denotes divergence between the products or product variants, whereas variability *in time* refers to product family evolution. According to Bosch et al. (2002), the differences among products are managed by delaying design decisions, thereby introducing variation points, which again are bound to a particular variant. The division into space and time variation also applies to quality variability.

A variation point identifies a location at which a variation can occur in a given system (Salicki & Farcet 2001). A variation point may be *external* or *internal* (van Gurp et al. 2001). External variation point refers to variability in the environment of the system, e.g. in peripherals, hardware platform or operating system, whereas internal variation point has to do with the internal variation of software functions or implementations. Similarly, quality properties may have internal and external variation points. An interesting characteristic of the quality property variation point is that variation points in function may cause *indirect variation* in the quality properties of software (Niemelä & Matinlassi 2005). For example, an external variation point in the execution platform may cause variability in software reliability and performance requirements i.e. the application has to compensate for varying reliability in platforms. Indirect variation may also occur the other way round i.e. variation in quality properties may cause variation in functionality (e.g. variability in security requirements results in different user authentication policies or varying data encryption algorithms).

In addition to variability in space and time, *discrete and continuous variability* are introduced by Becker et al. (2002). *Discrete variability* offers a set of possible features, from which a subset can be chosen for specific applications. Examples of variant subsets are represented in (Kang et al. 1990; Anastasopoulos & Gacek 2001; Bachmann & Bass 2001). These approaches introduce variant types (e.g. optional variant) stating the rules regarding how to select features in a variation point i.e. "select a function or not" and "select one of two alternative functions". However, these "yes/no" types of variants are not fully suitable to be used as quality property variants, which should also represent variation also in *quality property priorisation* (e.g. high/medium/low reliability requirements).

Continuous variability represents differing realizations, which can be parameterized later in the development process, namely in the compile, link or runtime phases (van Gurp et al. 2001). This conforms to the above mentioned idea of delayed design decisions and binding variants to variation points. Quality property variants are harder to bind later in the development process than functional variants. This is due to the fact that quality properties are realized by utilizing architectural styles and patterns in the development phase. However, dynamic architectures, see e.g. (Cheng et al. 2002), allow applications to reconfigure and evolve themselves at run-time e.g. through automatic updates of components. Figure 3 summarizes the variation types discussed here and illustrates the relationships between them. Variation dimensions are represented as axes and different types of variants can be placed in the variation space as is shown with examples in Figure 3.

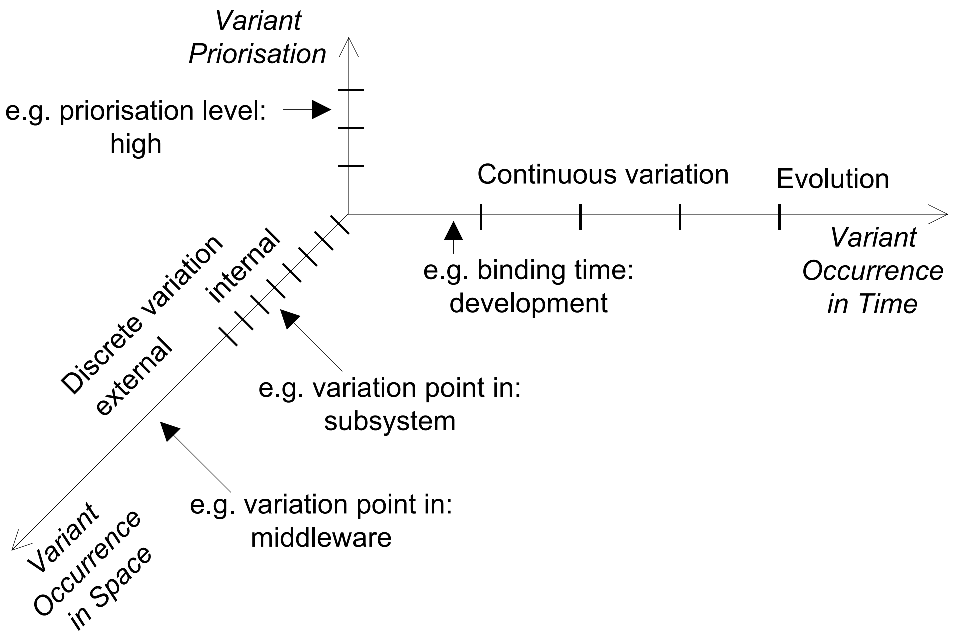

Variant Priorisation
e.g. priorisation level: high
Continuous variation
Evolution
Variant Occurrence in Time
e.g. binding time: development
Discrete variation
internal
external
e.g. variation point in: subsystem
e.g. variation point in: middleware
Variant Occurrence in Space

Figure 3. Quality variation space.

2.1.3 Quality representation in an architectural model

Architecture quality may be represented in a model, e.g., with quality profiles and with architectural styles and patterns. Quality profiles attach quality properties to a model whereas styles and patterns are used to *fulfil* the quality requirements in the software.

Quality profiles provide an informative way of mapping quality requirements and architecture, even for the purpose of automated or semi-automated architecture evaluation (Immonen & Niskanen 2005). Quality profiles are often implemented as UML profiles. A UML profile is a language extension mechanism that allows "metaclasses from existing metamodels to be extended to adapt them for different purposes" (OMG 2005b). That is, UML may be tailored, e.g., to model especially different platforms or domains. UML has already been extended especially to represent quality in the software model. Examples include a UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms (OMG 2003b), a UML Profile for Schedulability, Performance, and Time Specification (OMG 2003c), a reliability profile by Rodriques et al. (2004) and a quality profile for representing the reliability and availability requirements in architectural models by Immonen (2006).

Perry & Wolf (1992) define an architectural style expressing components and the relationships between them, with the constraints of their application, and the associated composition and design rules for their construction. Similarly, Bass et al. (1998) define an architectural style as a class of architectures and as an abstraction of a set of architectures that meet it. Further, architectural style supports the building of classes of architectures in a specific domain (Monroe et al. 1997). Architectural mechanisms realize architectural styles, thus promoting architecture quality.

A style is determined by a set of component types, a topological layout of the components, a set of semantic constraints and a set of connectors. In other words, architectural styles only describe overall structural frameworks for architectures and are not as much problem-oriented as architectural patterns. Architectural patterns are solutions for specific problems. Patterns are also widely reused and verified. Buschmann et al. (1996) divide patterns into *architectural patterns*, *design patterns and idioms*. Architectural patterns express the fundamental structural schema of a software system, and they are applied for high-level system subdivisions, distribution, interaction and adaptation. When an architectural style is strictly defined and commonly available, it can be regarded as a pattern (Niemelä 1999). Such are, for example, layered style (Klein & Kazman 1999) and layers pattern (Buschmann et al. 1996).

A design pattern (Gamma et al. 1994) describes a recurring structure of communicating components, which solves a general design problem in a particular context. Since design patterns are applied in a particular context, e.g. to define the content of a layer or a component, design patterns as such cannot guarantee a good overall architecture (Niemelä 1999). Idioms represent the lowest level of patterns, describing how particular aspects of components or relationships between them are implemented using a given programming language (e.g. Singleton for C++ language (Gamma et al. 1994)).

Although several studies have been made with goals to compare or categorize architectural styles – such as (Shaw 1995; Keshav & Gamble 1998; Levy & Losavio 1999) – there is still no common understanding on the subject. For example, there is no *explicit* list available on quality attributes and no explicit, common knowledge about which attribute(s) are promoted by the different styles. However, some remarkable research work has been done in this field, e.g., in (Andersson & Johnson 2001; Niemelä et al. 2005). Quite often, promoted quality attributes are derived indirectly from the style specification, e.g., Simplex uses redundancy to tolerate faults, thereby and therefore enhancing reliability. Furthermore, definitions vary. For example, publish-subscribe is known not only as a design pattern (Gamma et al. 1994) but also as an architectural style (Klein & Kazman 1999). Another example is the case where a model-view-controller architectural pattern (Buschmann et al. 1996) is also considered as a design pattern (Ardis et al. 2000).

Despite the confusion of definitions and specifications, I have collected some examples of architectural styles and the most important qualities they are claimed to promote in Table 6.

Table 6. Examples of architectural styles.

Name	**Quality properties**	**Reference(s)**
Layers style	portability, modifiability, reusability of layers	(Klein & Kazman 1999)
Software Architecture for Dependable and Evolvable Industrial Computing Systems, Simplex	redundancy, reliability	(Sha et al. 1995) and (Klein & Kazman 1999)
Chiron-2, C2	heterogeneity, concurrency, composition	(Taylor et al. 1996) and (Medvidovic et al. 1996)
Token architecture	(scalable) performance, extensibility, portability	(Karhinen et al. 1997)
Component programming architectural style, ComPAS	separation of concerns, locality	(Gall et al. 1997)
Dual protection style, DPS	security	(Fenkam et al. 2002)
Architectural style for deregulated power markets	flexibility, extensibility	(Zhao et al. 2001)
Point-to-point style	limits component accessibility	(Andersson & Johnson 2001)
Architectural style for end-user programming, E-Slate	increases end-user accessibility	(Birbilis et al. 2000)

2.2 Software architecture modelling

This section discusses the notion of software *model* and also introduces the area of *architectural model* descriptions. Further, this section introduces the concept of *model-driven software development* (MDD) and an initiative to standardise MDD, namely *model-driven architecture* initiative.

2.2.1 Software model

A model – as a design artefact – is an abstraction of the system from a certain point of view, wherein a system denotes a real world object. In other words, a design model is a representation of the real world and it represents the real world with a certain language (Bézivin 2004). According to another quite parallel definition “A model is a simplified representation of a system intended to enhance our ability to understand, predict and possibly control the behaviour of the system” (Neelamkavil 1987).

A model is written with a language, which may be textual or graphical (OMG 2005a) and the definition of a model shall not be dependent on the modelling language. In software modelling, such questions might posed as whether software is a real world object or is it a model of a real world object? Or if code is software or a textual model of software? Referring to Bran Selic in MDA summer school in September 2004, it may be stated that "Software has the rare property that it allows us to directly evolve models into full-fledged implementations without changing the engineering medium, tools and methods. The model evolves into the system it was modelling." The meaning of a model in software development may thus be concluded as follows: *everything is a model*.

According to Selic (2003), a *good* model is abstract, understandable, accurate, predictive and inexpensive. These criteria provide a kind of ordering to the chaos of software model definitions. An *abstract* model emphasizes important aspects while removing irrelevant ones. For example, in a textual software model (i.e. code) it is hard to emphasize anything and even harder to remove irrelevant aspects because a textual model can only provide inadequate mechanisms for abstracting. Järvinen (2001) refers to (Foley & van Dam 1982) by stating that the “human eye-brain pattern recognition mechanism does indeed allow us to perceive and process many types of data very rapidly and efficiently if the data are presented pictorially”. That is, graphical models are often faster to read and more efficient in representing things than text. However, the great power of graphical models may lead to a situation where a small mistake in making a graphical model causes a huge misunderstanding (Järvinen 2001). Therefore, the ‘abstract’ criterion sets high requirements for both the modelling language and the use of the language. The language shall provide mechanisms for abstracting away and the modeller shall be able to use these mechanisms correctly.

An *understandable* model denotes a model that is expressed in a form that can easily be understood by observers. Again, this criterion also sets a specific requirement for the modelling language. A software model has to be written in a language that is understood by other software developers. Formal modelling language is required to achieve universal understanding for both people and machines. Formal models are also required for interoperability of models and tools. However, formality in the sense of formal methods or mathematical formality is not a necessity in software modelling because mathematical software models (see e.g. Sifakis et al. 2003) are not understandable for most software developers. For example, if a building architect should represent his/her brilliant architecture as a mathematical model, it would not be easy to understand? In other words, *a model needs to speak the language of the reader in order to be understandable.* In an ideal situation, the appearance and language of a model should be tailored to the specific stakeholder(s). For example, it should be possible to represent different *views* of an architectural model to different stakeholders (see architectural views in Section 2.2.2 Architectural models). In addition to the modelling language, the modeller also contributes to making the model understandable: even the best language will not be understood if it is not used correctly.

An *accurate* model faithfully represents the modelled system. This criterion seems to contradict the first one: abstract and accurate at the same time? How is it possible to remove irrelevant aspects (i.e. abstract away) and simultaneously preserve the accuracy of a software model. Abstract is more of an aspect of the modelling language than it is an aspect of the model itself. In other words, the language should provide mechanisms for abstracting away, while this does not necessarily mean that different languages are needed for different abstraction levels. Accuracy has primarily to do with the model, i.e. how the modelling language is used to produce a model. Here, accurate models may be produced by drawing a model that includes more than one abstraction level. The upper levels of abstraction will remove irrelevant aspects, whereas the lower level(s) preserve accuracy. Such levels are represented by, e.g., user level, application level and technical level (Günther & Steenbergen 2004), and the conceptual and concrete levels in Paper II. This criterion sets requirements for the modelling method, in particular, and also requires support provided by a modelling tool. For instance, the method defines that a single language is to be used to describe several abstraction levels and the tool enables observation of the levels individually or at the same time.

A *predictive* model can be used to predict desired properties of a system from the model before a real system has been built. A predictive model also answers selected questions about the modelled system e.g., regarding how reliable the software is (Immonen 2006). However, one model is capable of representing the system "from a certain point of view", and thus one model cannot answer *all the questions* about the modelled system. Therefore, the question arises if we should talk about a software model or, rather, models? How many models are required to produce a *good* model as presented above? Only *one model* should be required to allow the information to be entered only once into the model. A model can also provide *multiple views* that represent the system from different points of view. In that sense, also the views are models as such, but if the information is entered only once, the entity can be considered as one model. In order to retain the benefits of a single model, all the views need to be consistent with each other while still separate to support the understandability and abstractive nature of a good model.

Finally, a model has to be *inexpensive*, i.e., much cheaper to construct and study than the modelled system. In software development, the most expensive thing are the people. The more people and working hours are required, the more expensive the software model and software itself. Thus, as a summary, a good software model reduces the amount of people and time needed in software development by being adequately abstract, understandable, accurate and predictive.

2.2.2 Architectural models

According to IEEE-1471 (2000), an architectural description aggregates one or more models and is organized by views, which again consist of one or more models. In other words, *an architectural description consists of models called views specified in viewpoints* (see definition on page 14).

An architectural view is a projection of the complete system model. Each architectural view/model represents selected parts of the system. Specific parts of the system may be targeted, for example, at specific stakeholders (e.g. customer, project manager, designer) or specific parts may represent only certain property/properties of the system (e.g. performance model).

There is no fixed set of architectural viewpoints, but viewpoints are rather defined by the method at hand (see e.g. (Hofmeister et al. 2000; Jaaksi et al. 1999; Kruchten 1995; Paper III). As defined in IEEE-1471 (2000), each architectural viewpoint may include one diagram or more, while a diagram is an element in a model. In the context of MDA, the modelling language (e.g. Unified Modelling Language 2.0, shortly UML2) may define certain guidelines for a set of diagrams used in models. However, in the end, the set of diagrams is specific for CASE tools and these tools rarely allow the user to configure any specific viewpoints or diagram sets. More specific information on the tools can be found in Paper I and in (Merilinna & Matinlassi 2004).

In addition to viewpoints, abstraction levels are peculiar to architectural models. Similarly to viewpoints, abstraction levels are not a fixed set either. Usually, a set of two or tree abstraction levels are used. The number of abstraction levels depends, e.g., on the size of the product/product family and application domain. Different abstraction levels may be used in architectural descriptions, for example: conceptual and concrete architecture (Paper II), commercial and technical (America et al. 2000) and user, application and technical architectures (Günther & Steenbergen 2004).

Summarizing the architectural models discussed here and the properties of a good model presented in Section 2.2.1, the use of views as architectural models promotes architectural models being *predictive*. Further, the use of abstraction levels assists in making architectural models *abstract* and *accurate*. Views and abstraction levels in company with an adequate modelling language make architectural models more *understandable*. The next section discusses an approach that enables architectural models – and software models in general – to be even more *inexpensive*: model-driven development.

2.2.3 Model-driven development

Model-driven software development, abbreviated as MDD (Selic 2003) or MDSD (Bettin 2005) focuses on providing models – rather than programs – as primary software products (Frankel 2003). In addition to the term "model-driven" also *model-based* (Törngren et al. 2005) and *model-centric* (Born et al. 2005) are used as synonyms to denote the approach defined above. Further, model-driven *engineering* is also used as a synonym for model-driven software *development*.

Adopting MDD in software development (especially in the domains of embedded and real-time software) faces hard resistance to change and, a change in large populations denotes great inertia. The code-centric approach is deeply rooted among software professionals, most of them thinking that models will never be accurate enough. Further, the model-driven approach arouses fear of loosing power. The power or the competence resides in people, and this generation of software engineers is grown to believe that the models are *in* the people, and thus, model-driven development is unlikely to be able to help this generation. In addition, MDD does not ensure any faster development process if the development work is done by a typical, relatively small and competent development team incrementally evolving their software product. The software culture has to be reconciled with MDD and with an appropriate motivator.

Hard resistance seems illogical considering that MDD embraces the principles of well-matured and industry-adopted software development trends. According to Bettin (2005), MDD is a multi-paradigm approach embracing the following trends:

• Domain analysis and software product line engineering
• Meta modelling and domain-specific modelling languages
• Model-driven generation
• Template languages
• Domain-driven framework design
• The principles of agile software development
• The development and use of Open Source infrastructure

In addition, applications of model-driven development already include large-scale systems, such as a billing and customer care system (Günther & Steenbergen 2004) and a distributed inventory tracking systems (Nechypurenko et al. 2004).

The next section provides an introduction to an initiative designed to standardize and therefore provide progress in the adoption of model-driven software development: model-driven architecture.

2.2.4 Model-driven architecture

Model-driven architecture, MDA (OMG 2003a; OMG 2005a) is an OMG initiative designed to provide a standardization framework for MDD. This framework comprises a set of non-proprietary standards that will specify interoperable technologies with which to realize model-driven development with automated transformations (Selic 2004). However, not all of these technologies will directly concern the transformations involved in MDA.

MDA Manifesto (Booch et al. 2005) introduces the three tenets of MDA:

- *Direct representation.* Direct representation means using models for representing problems rather than using models as graphical syntax for programming languages. The aim is to reduce the gap between domain-specific concepts and programming technologies. Models shall map directly to domain, not to computer technology. Direct representation reduces the effort required to implement complex applications.
- *Automation*. If models are used for representing problems, we will face a new semantic gap between the problem model and implementation technology. In order to fast up development and reduce errors, transformation over this semantic gap needs to be automated. Therefore, automation means using automated tools to transform domain-specific models into implementation code. This is the same thing that compilers do for traditional programming languages today.
- *Standards*. Standards are important because they promote the technology progress. Especially open standards ensure consistently implemented technologies, models etc. and openness encourages the adoption of standards by vendors.

In the MDA approach, a platform independent model (PIM) describes a system completely, without, however, any platform specific details. Platform specific details are described in a platform specific model (PSM), which is a realization of PIM. That is, MDA expresses model abstraction as *platform independence*. In the context of software development, it is important to define the platform. Frankel (2003) has proposed the following platform definition:

- Information-formatting technologies, such as XML (eXtensible Markup Language) DTD (Document Type Definition) and XML Schema
- 3rd and 4th generation languages, such as Java, C/C++ and Visual Basic
- Distributed component middleware, such as J2EE (Java 2 Platform Enterprise Edition), CORBA (Common Object Request Broker Architecture) and .NET
- Messaging middleware, such as WebSphere MQ (Message Queuing) Integrator and MSMQ (Microsoft Message Queuing).

MDA provides mechanisms for developers to capture their domain knowledge and to map it to implementation technology in a standardized form. This knowledge is used to produce tools that will hopefully do most of the low-level work automatically. "MDA has the potential to simplify the more challenging task of integrating existing applications and data with new systems that are developed" (Booch et al. 2005). Model transformation is the key concept in MDA. The next section considers architecture model transformation.

2.3 Architecture model transformation

The notion of model transformation is an essential element for MDA aiming at automated model transformation (see definition on page 14). Transformation *may be* bi-directional. Ramljak et al. (2003) introduce four different types of transformation: PIM to PIM, PIM to PSM, PSM to PSM and PSM to PIM. Christoph (2004) refines the definition of transformation by classifying transformations into two categories: horizontal and vertical. Figure 4 summarizes the relationships of the different types of transformations. It is hard to define a fixed number of PIMs and PSMs for software systems. For example, models may be written with various modelling languages, e.g. UML and ADLs (Architecture Description Language), resulting in several models at the same level of abstraction and for several platforms (e.g. Java and CORBA), which, again, results in several PIMs at slightly different level of abstraction.

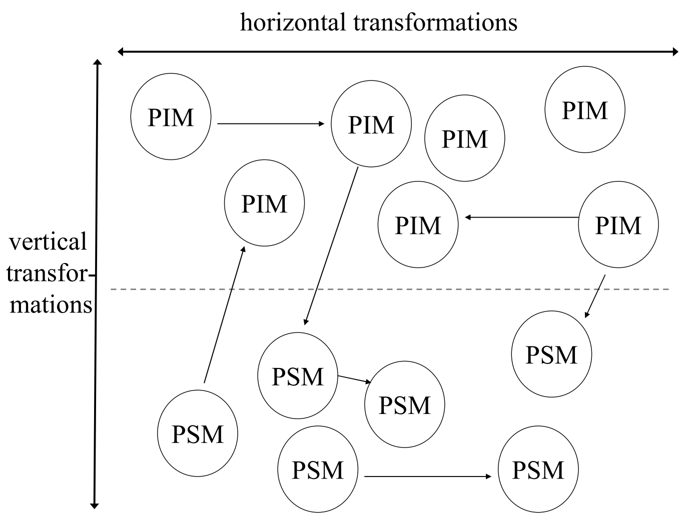


Figure 4. Different types of transformations.

In order to automate any type of transformation (which is the fundamental idea of MDA), the rules have to be written explicitly. The best way to define explicit transformation rules is to apply a standard transformation definition language for rule definition. The benefits of a specific standard transformation language are that (1) it is independent of the way of executing a transformation and, therefore, it enables automation with any other language, e.g. a procedural language, and (2) it is unambiguous and commonly understood.

At the moment (November 2005), the standardisation of a common transformation language is still underway. The OMG Query/Views/Transformation request for proposals (OMG 2002) was announced in April 2002 and initial submissions were due on October 28th, 2002. A total of eight submissions were received as proposals (Gardner et al. 2003). OMG is currently finalising the standard. The final competing proposals are the QVT (Query/Views/Transformations) Merge approach and the QVT Compuware/Sun approach (Grønmo et al. 2005). The QVT standard will be a general purpose language for model-to-model transformations.

Standardization of model-to-text transformations is an ongoing process within OMG. The OMG MOF (Meta Object Facility) Model to Text Transformation request for proposals (OMG 2004) was announced in August 2004. Initial submissions were due January 10th 2005 and the issue is in progress.

In addition to using a special, dedicated transformation language, model transformation can be defined through other approaches as well. Czarnecky and Helsen (2003) provide a domain analysis for different transformation approaches – including five QVT proposals. Sendall & Kozaczynski (2003) have proposed two different approaches for defining transformations: (1) direct model manipulation and (2) intermediate representation.

Direct model manipulation refers to accessing the model representation and the ability to manipulate the representation. In practice, the direct model manipulation approach relies largely on modelling tool properties. The modelling tool may provide a set of procedural APIs (Application Programming Interfaces) for manipulating the model, while the API is accessed with a general-purpose language such as Visual Basic or Java.

Intermediate representation also requires tool support. The modelling tool may support exporting the model in a standard form (e.g. XML), so that it can be transformed with an external tool and then imported back to the modelling tool.

I may draw the conclusion that a dedicated transformation language is a language-oriented approach for defining transformations, whereas direct model manipulation and intermediate representation are tool-oriented approaches. The tool-oriented approaches suffer from several disadvantages as compared with the language-oriented approach. Such flaws are, for example, that they are not as expressive as dedicated language and that they provide automation only through a specific tool or tools. While the language-oriented approach is tool independent, it does require becoming mature enough before it can be widely used. Grønmo et al. (2005) evaluated the two competing transformation language proposals. Some of the highlights of their report are summarized in Table 7.

Table 7. Summary of the QVT language proposal evaluation.

Language proposal	Advantages +	Disadvantages -
QVT-Merge	• Graphical syntax can define single transformations fully graphically (in some complex transformations OCL (Object Constraint Language) annotations are needed) • Easy to learn: - textual language shares many similarities of both syntax and constructions with well-known object oriented languages such as Java, c# and c++ - graphical notation is quite intuitive to understand	• Graphical syntax not complete - lack of graphically specifying compositions such as "parallel split" and "synchronization" • Difficult to learn: - Ambiguous guidance on how to use the language - Many implicit constructions for shorthand notations
QVT Compuware/ Sun	• Easy to learn: - concise specification - UML, MOF and OCL reused with very few extensions	• Graphical syntax not complete - graphical notation cannot be used to fully define any transformation that can be defined textually - unclear on how to define multiple target models • Violates the evaluation criteria: - no support for traceability - no support for black-box interoperability - no support for composition of transformations • Difficult to learn: - lack of examples and explanation of some of the syntax used

Both languages offer a complete textual syntax for describing transformations between any two MOF models. Both languages have disabilities concerning graphical syntax (see Table 7 for details). Based on the test users opinions in eight example transformations, the average ease of use score for the QVT Merge language was approximately 2.5 (maximum 5) and for QVT Compuware/Sun the score was 3 in one example transformation. Despite the lack of examples in the evaluation, the QVT Compuware/Sun language proposal can be considered easier to learn than QVT Merge (Table 7). On the other hand, the QVT Compuware/Sun approach has several disadvantages that even violate the evaluation criteria set for QVT proposals.

The QVT language is defined and almost standardized, but no QVT compliant tools (e.g. syntax parser) exists yet (Grønmo et al. 2005). Although standardized transformation language is not yet supported by any tool, various other kinds of model transformations (Czarnecky & Helsen 2003) are supported in several tools. OMG lists 55 tools on the page of MDA committed companies and their products[3]. In addition, modelbased.net[4] (a web site dedicated to tools and information related to model-driven software development) mentions 13 open source tools just for MDA transformation. Especially the Eclipse[5] tool has evolved into "a rich software ecosystem that has spawned an active open source community" (Frankel 2005).

[3] http://www.omg.org/mda/committed-products.htm
[4] http://www.modelbased.net
[5] http://www.eclipse.org/

3. Towards automation of quality-driven architecture model transformation

This section expands the original idea of QAMT presented as a short paper (Matinlassi 2005). The section is structured as follows. First, an overview is provided on how to approach QAMT automation. The overview describes the main activities of QAMT and introduces the actors collaborating in transformation. Second, model activities are refined to show how different parts of the model are automated and how the automation actors operate in the transformation process. Third, two automation actors – stylebase and rulebase – are discussed in more detail.

3.1 Introduction to QAMT automation

Quality-driven software architecture transformation requires intellectual and complex reasoning carried out by humans. In order to automate processing, the complex reasoning needs to be simplified. Therefore, making *automation* of QAMT possible requires developing a model that describes (simplified) *manual* QAMT. Manual QAMT is then further divided into more detailed activities and automation is approached by automating the individual activities. The QAMT model is described with activity graphs according to UML2 (OMG 2005b).

Figure 5 illustrates the top-level activity graph for *manual* QAMT. An *input pin* (rectangle) serves as a transformation trigger for QAMT *activity* (rounded rectangle). Quality variability as a transformation trigger (see Section 2.1.2 Variability in quality properties) makes transformation quality-driven. In transformation, an architect identifies source and target and then converts source model into target model.

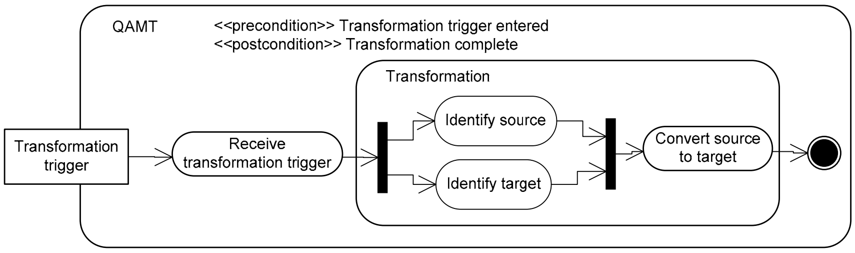

QAMT
<<precondition>> Transformation trigger entered
<<postcondition>> Transformation complete
Transformation
Transformation trigger
Receive transformation trigger
Identify source
Identify target
Convert source to target

Figure 5. An overview of the quality-driven architecture model transformation.

Figure 6 refines quality-driven architecture model transformation. I will later (in Section 3.2) concentrate on the automation of the activities marked with grey colour. *Identify source* denotes identifying the potential parts of the architecture that require modifications in order to meet the requirements set for the model. Therefore, the architect needs to carefully study the existing architecture model, to evaluate architecture against the new quality requirements and to select the parts of architecture that would be influenced in transformation. Quality evaluation may require using special architecture analysis methods, such as introduced in (Dobrica & Niemelä 2002). *Identify target* is about finding out how the source will be changed in transformation. The target architecture model may have new, removed or modified components and connectors. Identifying the target often requires searching for and studying several alternative target models before making the final decision. The *Convert source to target* activity in Figure 6 illustrates updating an architectural model manually.

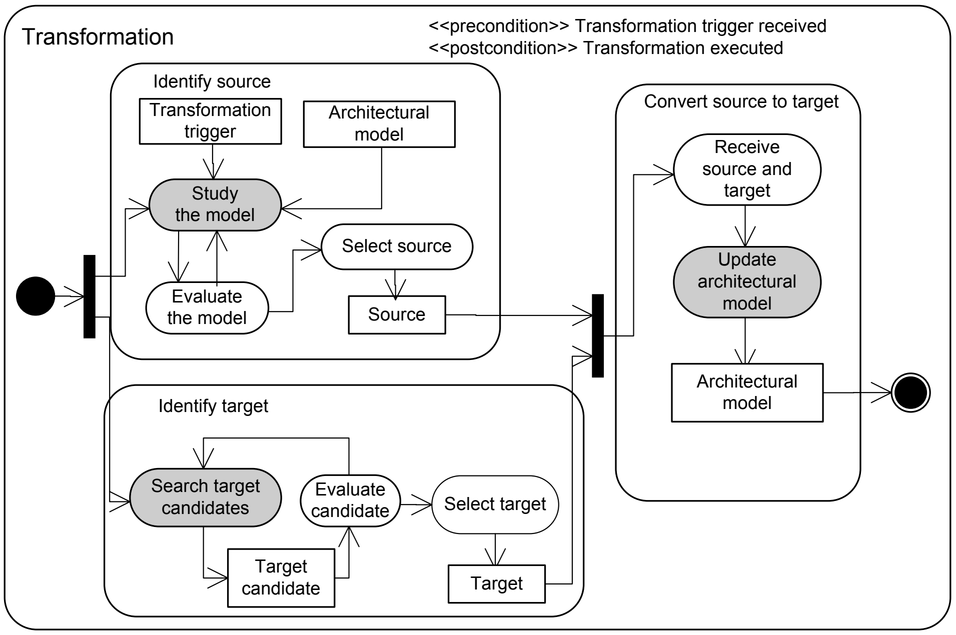


Figure 6. Manual transformation.

Next, the communicating actors comprising the QAMT model are defined in a collaboration diagram (OMG 2005b) in Figure 7. The communicating actors are: architect, modelling tool, modelling tool extension, stylebase and rulebase. Architect and modelling tool are the fundamental actors needed in QAMT, whereas stylebase, rulebase and modelling tool extension are so-called additional actors added to the model for the sake of automation. The automation actors can be applied not only in executing the transformation but also in providing automated guidance for the architect in selecting source and target patterns. Especially the decision-making process in selecting target patterns might be tricky without automation, which will enable an easy way of trying out various approaches to problems.

“Architect” represents the person(s) responsible for transforming the architecture. “Modelling tool” is a CASE tool including the software architecture model, which is described according to the architecture description principles of the QADA methodology (Papers I–VII) with UML2 language. The methodology defines up to four viewpoints to software architecture. The selected viewpoints with included diagrams are modelled in a CASE tool. Although the features supported by commercial modelling tools vary, it is supposed that these modelling tools do not include such advanced features as automated QAMT. The QAMT specific features that are not present in the commercial CASE tool are represented with a "Modelling tool extension" actor. These specific features include such features as a user interface for stylebase and automated transformation. An implementation of the modelling tool extension is presented in (Merilinna 2005).

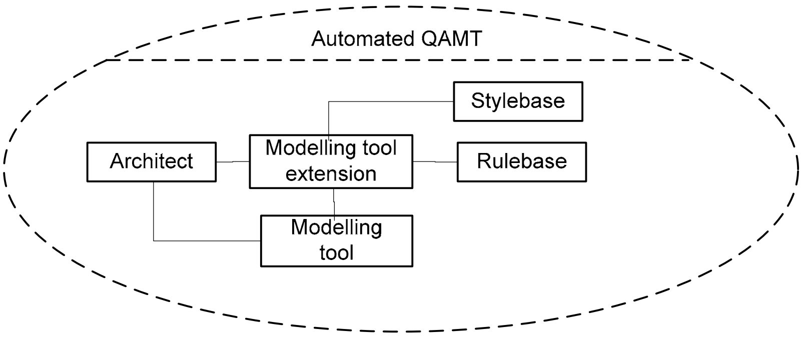


Figure 7. Collaboration diagram for automated QAMT.

"Stylebase" is a knowledge repository where architectural patterns (see Section 2.1.3 Quality representation in an architectural model) are stored in a uniform way. Stylebase is used for recording, managing and utilizing architectural quality solutions in order to promote automation. The data stored in the stylebase is strictly defined and it includes commonly available styles, i.e. architectural patterns. Although the name “stylebase” of this knowledge base may appear slightly misleading; it is called a stylebase because “patternbase” could easily be confused with design patterns.

"Rulebase" is a knowledge repository where transformation *mapping* is presented with *rules* (see definitions on page 15). Each transformation rule defines a specific transformation from architectural pattern A to architectural pattern B.

Table 8 maps QAMT activities (illustrated with grey colour in Figure 6) and actors (Figure 7) together. In the next section I further clarify how collaboration is done. An automation model is used to combine the manual QAMT model and the collaboration model towards automated QAMT.

Table 8. The selected QAMT activities and automation actors that collaborate in the automation of each activity.

QAMT activity	**Collaborating QAMT automation entities**				
	Architect	Modelling tool	Mod. tool extension	Stylebase	Rulebase
Study the model	√	√	√	√	
Search target candidates	√		√	√	
Update architectural model	√	√	√		√

3.2 QAMT automation model – activities

Figure 8 illustrates a model used for the automation of the first activity in QAMT (i.e. Identify source). The activity diagram is categorized upon collaboration actors in Figure 7: architect, modelling tool extension, modelling tool, stylebase and rulebase. The architect studies and evaluates the source model against the quality requirements set for the target model. This evaluation step in the process is semi-automatic. An architect may, for example, search for all the architectural patterns utilized in the source model. The search may also be constrained to:

- Architectural patterns in the source model supporting a specific attribute, e.g. reliability
- Architectural/design patterns expressed only in certain diagrams (e.g. structural or allocation diagrams) of the source model.

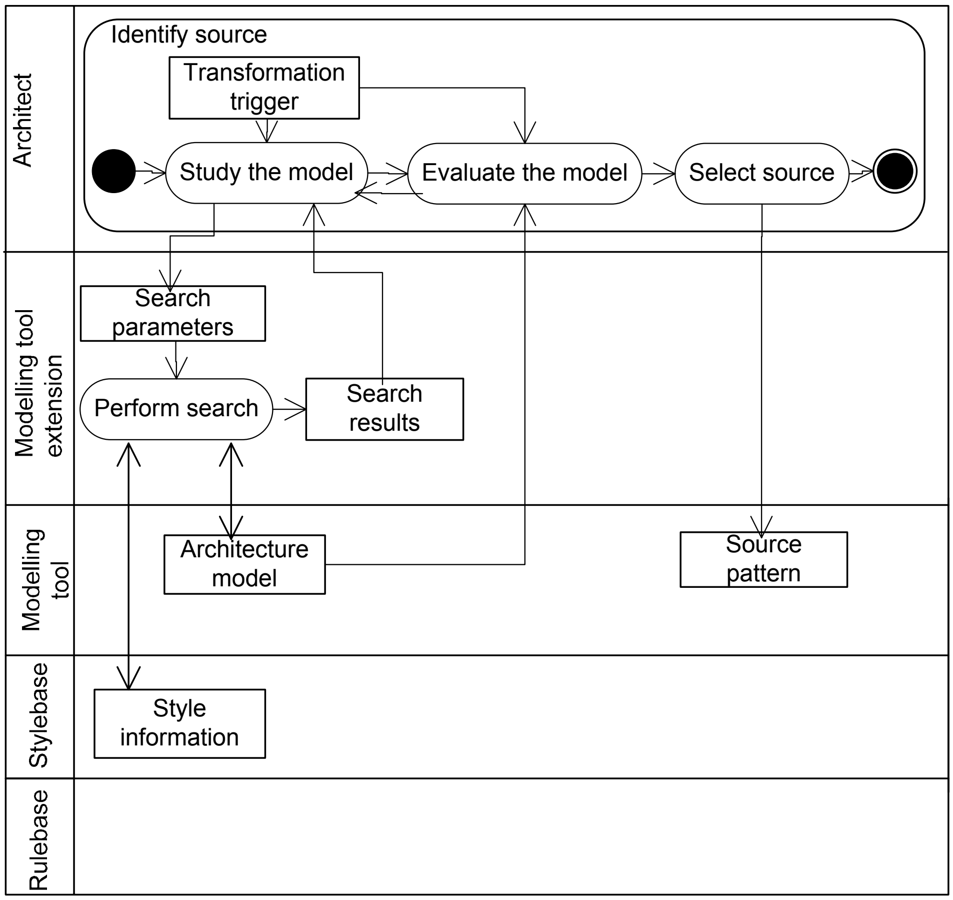


Figure 8. Automation model for the "Identify source pattern" activity.

As the result of this step, the architect identifies potential parts of the architecture that require transformation in order to meet the requirements set for the target model. In order to pinpoint a pattern in a model, the following information – marks – are required in some form for each component in the architectural model. Marks utilize stylebase parameters, which are defined later (Section 3.3.1 Stylebase).

1. What is the type of the component? Component type is defined because only components with same type (e.g. data component) can be reused.
2. What pattern(s) does a component contribute to? One component may be used in one or more architectural styles. The component needs to contain a reference or references to the stylebase.
3. What is the role of the component in the pattern? Role defines component behaviour quite extensively and the role information is needed for reusing component behaviour.

In view of the fact that quality properties are not used as marks, evolution qualities and execution qualities should be attached to the architectural model. This is not due to transformation capabilities but rather to semi-automatic model evaluation. Quality requirements attached to the architectural model will make software architecture evaluation significantly easier. The attaching of quality properties may be done with special UML2 quality profiles (see Section 2.1.3 Quality representation in an architectural model).

In the second activity, the architect identifies potential candidates for target model architecture. Figure 9 presents a model towards automation of the "Identify target pattern" activity. In the same way as searching the model in the previous activity, the architect may search the stylebase directly, for instance, for the following:

- Are there any allocation styles available that support modifiability? Allocation styles are examples of styles that are visible at least in the architectural deployment viewpoint.
- What style(s) would be suitable for the problems of extensible architecture? A search with only one search parameter: quality attribute.
- Is it allowed to transform the style found in the model into something else? A style guide would assist the architect in utilizing a pattern.

The architect makes the decision about transforming the model. Here, the architect selects the pattern or patterns requiring to be transformed while also making the decision for suitable target patterns. Although the decision is guided by the information available in the tool and in the stylebase, the architect is, in the end, responsible for the final decision.

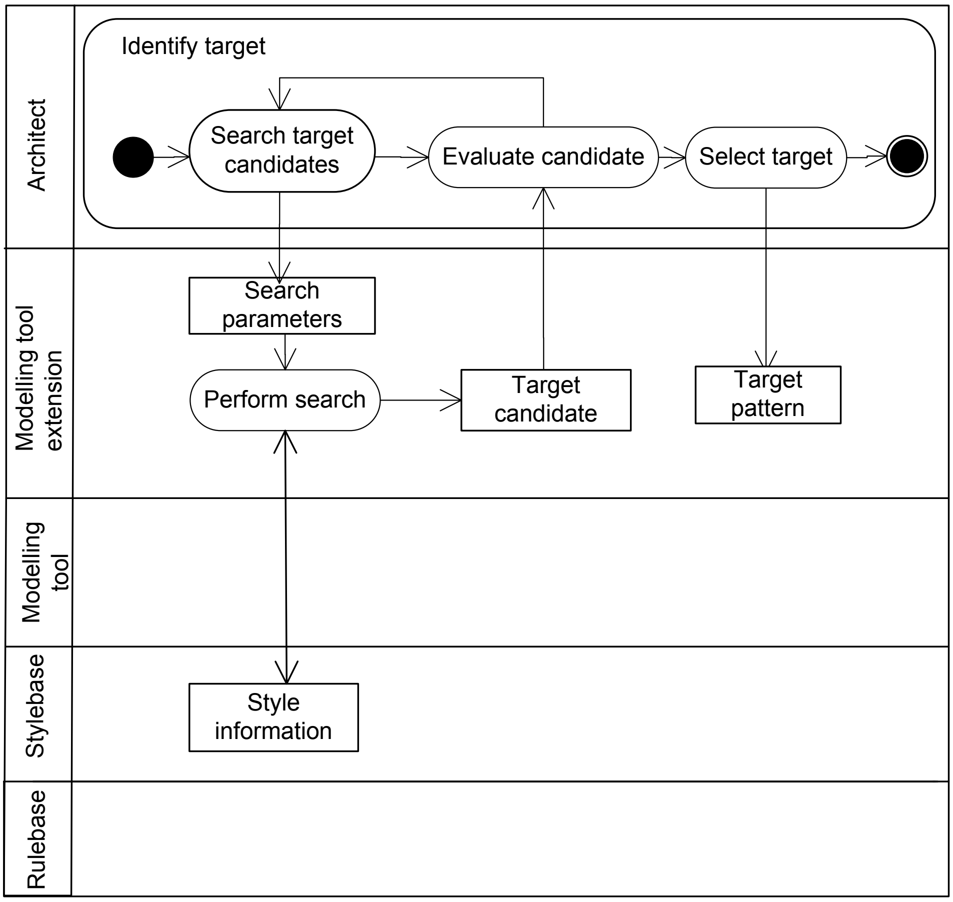


Figure 9. Automation model for "Identify target pattern" activity.

After the architect has made the decision, the transformation can be performed. Figure 10 presents a model towards automation of the "convert source to target" activity. Transformation rules are applied to convert source to target. Definition of transformation rules is presented later, in Section 3.3.2 Rulebase. In addition to employing transformation rules, the tool uses source pattern data and target pattern data in the automation of the transformation. Source pattern is often a special instance of a pattern (e.g. the number of layers is not predefined in the layers pattern) and therefore, the tool must utilize source pattern information in the model, not only data in the stylebase. Finally, the architect possibly needs to implement new connectors in the target model, i.e. to define how the transformed part of the architecture connects with the remaining architecture.

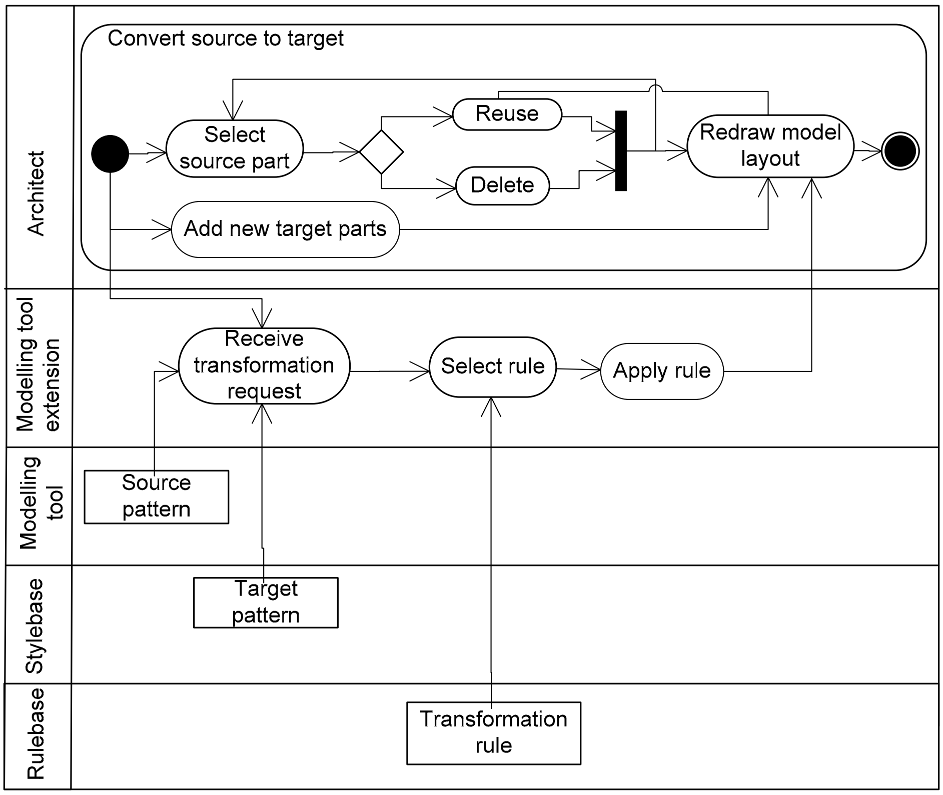


Figure 10. Automation model for the "Convert source to target" activity.

3.3 QAMT automation model – actors

In this section, two of the QAMT automation actors are presented: stylebase and rulebase. The third automation actor, modelling tool extension, is discussed in (Merilinna 2005). This section discusses the following actor properties:

- What information shall an actor contain and why? That is, what information shall be included in the stylebase and in the rulebase. The stylebase is designed for use as an automation actor for all transformation activities (i.e. selecting source pattern, selecting target pattern and converting the model) whereas the rulebase is mainly designed for use as an automation actor for transforming the model. The information included in the actor shall serve the use which the entity is designed for.

- How to capture information in a uniform way? The information needs to be captured and represented in a way allowing it to be easily and without misunderstandings translated into implementation. That is, the information for stylebase is defined with parameters and predefined parameter values. The mapping in the rulebase is defined with natural language.
- How to obtain information? Obtaining actor information may be complicated because of diverging parameter value definitions and representations. Especially for the rulebase, a technique for defining new rules is introduced.

Although this section does not define how to implement actors or how to represent information included in the entities, it does suggest some examples of these.

3.3.1 Stylebase

Table 9 illustrates the structure of the stylebase data. Although several attempts have been made to categorize architectural styles and patterns (see Section 2.1.3 Quality representation in an architectural model), none of them was found suitable as such for describing styles in a stylebase. Pattern name and component type parameters are based on the architectural style catalogue format (Shaw & Clements 1996; Shaw & Clements 1997), whereas the remaining nine parameters are specially defined here to complement the format so as to make it support transformation better. The table also discusses the relevance of each parameter and gives some examples on how parameter information may be implemented.

The first two stylebase parameters – *pattern name and reference* – are required for identifying a pattern. In the literature, there are several definitions for a single pattern name, or, one commonly known pattern may have several different names. Therefore, two parameters are needed. For example: "Layered pattern according to Buschmann et al. 1996" gives a pattern a unique identifier and a reference for its definition.

Table 9. Stylebase data parameters.

Parameter	**Relevance**	**Example implementation**
Name of pattern	Identification of a pattern	Model/diagram name
Reference		
Definition	Defining pattern structure, behaviour, component and connector layout according to *reference*	Textual pattern description embedded in model documentation
Figure		Structural template model of the pattern
Quality attribute	Mapping requirements and patterns	Qualities promoted by a pattern presented in a table
Rationale		Quality note
Component type(s)	Defining pattern structure and behaviour	Component stereotype
Component role(s)		Behaviour template model of the pattern
Abstraction level	Selecting admissible transformations	Embedded in model, e.g. as diagram documentation
Purpose		
Diagram		Tool specific diagram name

The next two parameters, pattern *figure* and *definition,* are included for the convenience of the end user. Figure serves for illustrating pattern layout, while definition includes the information and tips on how pattern may be utilized to its full capacity. The figure illustrates the layout, i.e. topology, of pattern components. Topology describes the geometric shape that the data or control take in the architecture (Shaw & Clements 1997). For example, a layered style has a hierarchical control topology (control passes from upper layers to lower layers) and blackboard has a star control topology (central control component invokes surrounding data components). The values for these parameters are not predefined and they mainly serve the purpose of the semi-automatic transformation activity "identify target".

The next two parameters – *quality attribute and rationale* – are essential for mapping quality requirements with a pattern. The quality attribute reveals the software qualities promoted by the pattern. The *quality attribute* parameter is complemented by *rationale*. Since the interdependencies between patterns and quality attributes often are complicated and implicit, the rationale is recorded in order to clarify the mapping between qualities and patterns. For example, "The Simplex pattern promotes reliability through tolerating software faults and providing a redundancy mechanism".

Pattern behaviour is defined with the parameters of *component type and component role*. Software components may express several different types. The list of component types introduced here is based on the experience gained in cases C1–C5 (Papers IV, V, VI and VII). So far, five main types of component have been identified: data, control, computation, package and interface (Table 10). *Component role* refers to pattern description in *reference* and describes what the responsibilities of a component in a pattern are. Therefore, roles provide a predefined description of component behaviour. For example, "computation component represents a client role in the blackboard pattern". Referring to the blackboard pattern definition we may sketch the behaviour for the different components.

Table 10. Summary of component types in stylebase.

Comp. Type	**Description**
Data component	Examples of data components are files, databases and data structures. Most often the data component is passive and accessed by other (typically computation) components.
Control component	Control components have the property of mastering other components by invoking them or controlling their access rights.
Computation component	Computation components typically process data, i.e. they acquire input data, process it further and finally produce output data. Algorithms are examples of computation components.
Package component	Package components are used to categorize other components and they do not include any functionality. However, package components may involve non-functional requirements which are common to all components inside the packaging element.
Interface/port component	Interfaces neither contain data nor compute, but they provide an access to the components behind the interface and may also hide the implementation of the accessed components.

Information about the pattern *abstraction level*, *purpose* and *diagram* is required for selecting admissible transformations. For abstraction levels, we apply the definition of Buschmann et al (1996): architectural patterns, design patterns and idioms. Since idioms, however, are dependent on programming language, they are out of the scope of this study. Transformations are allowed only between patterns at same abstraction level. Thus, for example, an architectural pattern cannot be transformed into a design pattern. Similarly, transformable patterns have to have the same *purpose* and *diagram*.

For the *purpose* of the pattern, there exist several categorizations. Gamma et al. (1994), for example, define five different purposes for design patterns: structural, behavioural, fundamental, concurrency and creational. The purpose of architectural patterns is roughly divided into four categories (Buschmann et al. 1996): interactive systems, from-mud-to-structure, distributed systems and adaptable systems. The categories stated above are applied for design patterns and architectural patterns as well. The set of patterns mentioned here is not an exhaustive one, and when new patterns are added, new categories may be required. For example, if a special pattern supporting reliability is added to the set, the architectural pattern categories are complemented by a new purpose: tolerating software faults.

3.3.2 Rulebase

The rulebase shall include the information relevant for mapping in architectural model transformation. That is, a rule definition technique is needed to create the rules on how to transform particular source patterns to corresponding target patterns. When defining rules, the constraints for transformations need to be defined first. Architectural transformations are admissible between patterns that

- are effective at the same *abstraction level*
- have the same *purpose* and
- are illustrated in a similar type of *diagram*.

Respecting the constraints above, the activities for defining a new transformation mapping rule are introduced in Figure 11. The goal of these activities is to create the target pattern by reusing the properties of the source pattern to as great an extent as possible. The precondition for applying the technique is that the architecting process for the structure of the target architecture is defined. For example, the main steps of how to build the structure of a blackboard pattern are: define the blackboard component (i.e. the central data store), specify the control component and implement the knowledge source components (knowledge source components process data placed in the database).

Target pattern components are implemented by analysing the target pattern against the source pattern, paying attention to the following points: (1) Do source and target have the same types of components, i.e. what building blocks/components can be reused? An architectural pattern mapping rule is a source pattern analogy with the target pattern. Pattern analogies are not always simple one-to-one correspondences but the equivalency is often one-to-many or even many-to-many. Therefore, a rule may need to be defined as an *interactive* rule. An interactive rule interacts with the user while choosing the target solution. That is, mapping rules may have variation points. (2) Do we have to create new components? New components are not necessarily completely new. Creating new component means creating a skeleton component with a default role behaviour rather than reusing the existing component behaviour from the source pattern. Typically, a control component is often pattern specific and may not be reused, whereas computation and data components are defined by the user and therefore have more reuse potential.

The mapping record between the source and target components is monitored. For each component, the same types of data are recorded: the pattern name that the component contributes to, the role the component represents in the pattern and the component type. The mapping record serves for reversing the transformation. However, not all the mappings are reversible without transformation record because of one-to-many and many-to-many equivalencies.

The “Implement components” activity is repeated until all target components are implemented. Next, new component topology is implemented for the target model. A component topology is predefined for architectural patterns. However, refinements may need to be done to the default topology after the transformation. This activity may also result in creating an interactive variation point into the mapping rule. Finally, connectors are implemented. If the source connectors include defined messaging protocols or other reusable functionalities, the source connectors may be reused.

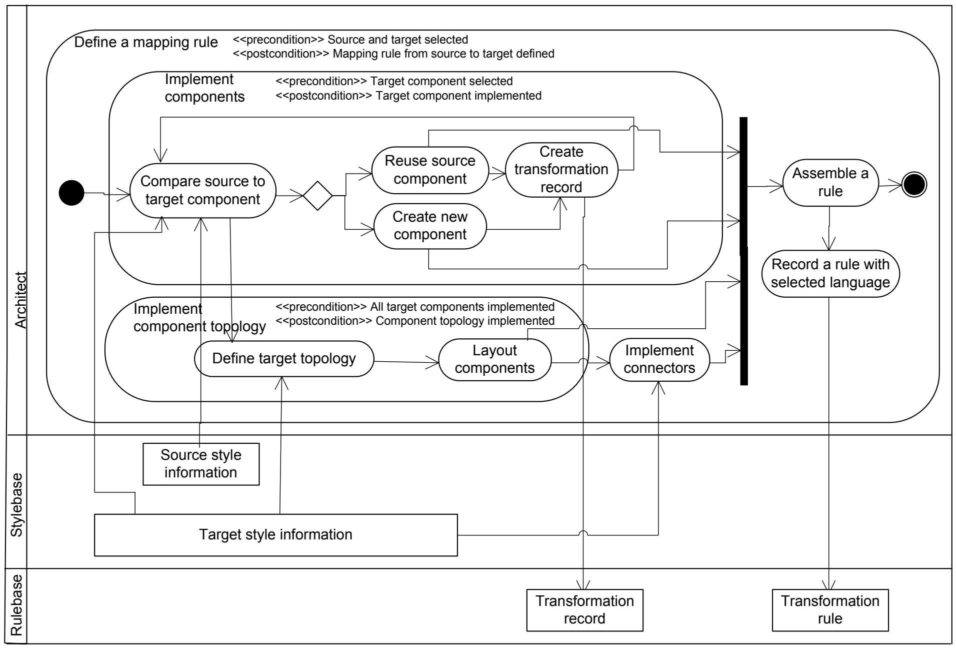

Architect
Stylebase
Rulebase
Define a mapping rule
<<precondition>> Source and target selected
<<postcondition>> Mapping rule from source to target defined
Implement components
<<precondition>> Target component selected
<<postcondition>> Target component implemented
Compare source to target component
Reuse source component
Create new component
Create transformation record
Assemble a rule
Record a rule with selected language
Implement component topology
<<precondition>> All target components implemented
<<postcondition>> Component topology implemented
Define target topology
Layout components
Implement connectors
Source style information
Target style information
Transformation record
Transformation rule

Figure 11. The activities of the mapping rule definition technique.

Assembling a mapping rule requires information from several activities of the technique. The rule is recorded with a selected language (see possible options introduced in Section 2.3) and saved in the rulebase.

4. Evaluation of the QAMT automation model

Evaluation of a design artefact – according to March and Smith (1995) – involves (1) developing criteria and (2) assessing artefact performance against those criteria. This section presents what the goals are that the QAMT model tries to accomplish and how well it succeeds in accomplishing the goals. The goals are represented as metrics for assessing the model (Section 4.1) and the assessment results reveal how well the model works (Section 4.2). A summary (Section 4.3) will provide the assessment results in a nutshell.

4.1 Deriving the goals

The goals of a model have their origin in the needs of the stakeholders. Four main stakeholders for the QAMT model can be recognised: mapping language developer, transformation developer, tool developer and end user (Figure 12). Each stakeholder deals with the QAMT model from a different point of view and therefore has different interests concerning the model.

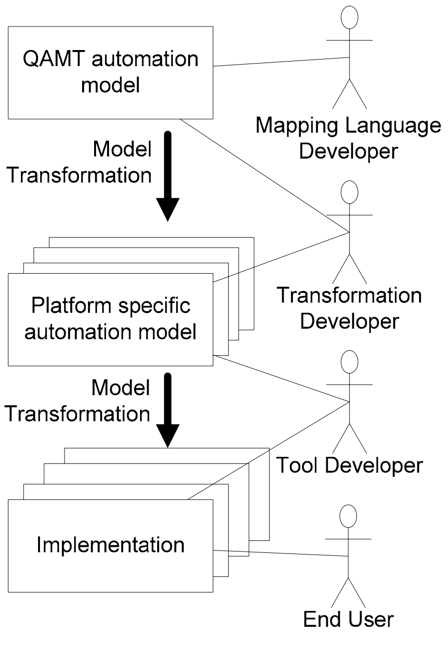


Figure 12. Stakeholders of the QAMT model.

The *mapping language developer* translates the QAMT mapping from one mapping language to another mapping language. The mapping language may be a natural language, an action language (an algorithm), or a model mapping language (see Section 2.3). This stakeholder is mainly interested in the mapping part of the model. No matter what the language is, the mapping needs to be completely defined to allow it to be translated into another language.

The *transformation developer* transforms the QAMT model into a specific implementation platform. This stakeholder is interested in one part of the model at a time. A platform specific model defines, for example, what technologies are used for implementing stylebase and in what form the data is represented. In addition, the platform specific model may define what commercial or open source tools are used as a platform for implementation or if the tool is developed from scratch. Therefore, the transformation developer appreciates platform independence for all the parts of the QAMT model. Platform dependency would restrict the work of transformation developer. On the other hand, conformance to certain standards/practices, e.g. modelling language and transformation specification, will help the work of transformation developer.

The *tool developer* implements the QAMT platform specific model in a tool. The implementation further refines the platform specific QAMT model by adding visual representation, usability issues etc. to the model elements. The tool developer does not directly deal with the QAMT automation model but rather with a platform specific definition of QAMT and, therefore, the *tool developer* interacts with the *transformation developer*. The tool developer appreciates platform independence (tools, programming languages) and model maturity.

The *end user* uses QAMT implemented in a tool. The end user is usually a software architect that wishing to perform a quality-driven software architecture model transformation automatically. Although the end user deals directly with the implementation – which is only a single instance of the QAMT model – the end user does have some interest concerning the QAMT model. The model shall provide a high level of automation for the end user, and, in order to achieve automated transformation, the transformed model shall utilize a MOF compatible modelling language (Selic 2004). For a comfortable use experience, the end user requires style descriptions that are informative and as complete as possible.

Each stakeholder has different interests concerning the QAMT model. The stakeholder requirements are grouped into form goals and a summary is presented as an evaluation framework in Table 11. Among these goals, *Completeness* denotes how complete the transformation specification is in terms of the OMG MDA transformation definition. Transformation inputs shall be presented as a marked model, while the mapping and outputs shall take the form of a transformed model with a record. Since the stylebase includes valuable knowledge utilized in all activities of the transformation, stylebase completeness is evaluated separately. The completeness criterion deals with model *actors* similarly to the goal of *platform independence*. Platform independence measures how platform independent the model really is. The next two goals are related to model *activities*. *Level of automation* measures whether the activities of the model are manual, semi-automatic or automatic. *Maturity* reflects the empirical validation of model activities. Lastly, *Conformance to standards* lists the most important standards related to transformations, modelling and quality, and assesses whether or not these are supported by the model.

Table 11. Evaluation framework for the QAMT model.

Goal	**Requirement(s)**	**Stakeholder(s)**
Completeness	Complete specification for mapping	Mapping language developer
	Complete specification for marks and record	Transformation developer
	Complete definitions for styles	End user
Platform independence	Portable to many technologies and implementations	Transformation developer
	Independent of tools, programming languages	Tool developer
Level of automation	High level of automation	End user
Maturity	Is the model validated with empirical data?	All
Conformance to standards	Modelling language MOF compatible	End user, Transformation developer
	Transformation specification conforms to MDA	Transformation developer
	Terminology follows state-of-practice	All

4.2 Assessment

This section evaluates the QAMT model against the evaluation criteria presented above. Table 12 illustrates how the evaluation criteria focus on the different aspects of the QAMT model.

Table 12. Concerns of evaluation criteria.

Criterion	**Concerns**		
	Activities	**Actors**	**Model**
Completeness		√	
Platform independence		√	
Level of automation	√		
Maturity	√		
Conformance to standards			√

Each criterion is discussed in a separate subsection, which provides a more detailed definition of the criterion in question and an evaluation against the criterion.

4.2.1 Completeness

Figure 13 illustrates an overview of a transformation. The architect takes a source model, marks it and then the marked source model is used to prepare the target model according to mapping. The transformation can be done manually, with computer assistance, or automatically. The transformation produces a target model and a transformation record, which traces the transformation back to the source model (OMG 2003a).

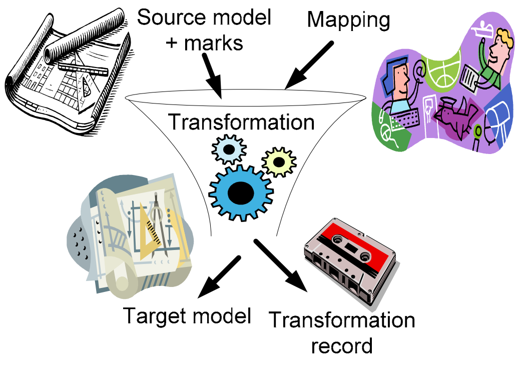


Figure 13. An overview of transformation.

Table 13 and Table 14 present an evaluation of QAMT completeness. In Table 13, the transformation elements are those defined in the MDA guide (OMG 2003a), excluding source and target models, which are provided for transformation and not considered as part of the QAMT model. QAMT includes two main automation actors: stylebase and rulebase. The rulebase corresponds directly to mapping, whereas the stylebase as such can not be considered a transformation element but it rather provides assistance for all the transformation elements. The stylebase assists in developing the source and target models by providing knowledge on existing modelling patterns. Source model components are marked with stylebase parameters. The definition of mapping rules also utilizes stylebase knowledge and the record tracks transformation according to stylebase data. Further, the stylebase is an automation actor which has a remarkable impact on end use convenience. From these points of view, stylebase completeness is evaluated here as a part of marks, mapping and record (Table 13) and also as a separate automation actor in Table 14. Table 14 illustrates the support of stylebase parameters for transformation (marking, mapping, recording) and for utilizing stylebase as an architect's handbook while designing and evaluating an architecture model.

Table 13. Completeness of transformation elements in QAMT.

Element	**Completeness**
Marks	Marks are supported by associating following stylebase data parameters to the modelling components: component type, style and role (see p. 55)
Mapping	Mapping is supported in the rulebase, but not completely. The rulebase includes natural language rules for constraining transformations and a technique for defining new mapping rules (see p. 61). The mapping rule definition technique utilizes the data parameters of the stylebase.
Record	The generation of transformation record is supported in the mapping rule definition technique (see p. 62). The record traces target model component mapping back to the source model with marks (stylebase parameters).

Table 14. Evaluation of stylebase parameters.

Stylebase parameter	**Supports**			
	Marking	**Mapping**	**Recording**	**Using**
Name of pattern	√	√	√	√
Reference		√		√
Definition				√
Figure				√
Quality attribute				√
Rationale				√
Component type(s)	√	√	√	√
Component role(s)	√	√	√	√
Abstraction level		√		√
Purpose		√		√
Diagram		√		√

4.2.2 Platform independence

Platform independence (see Section 2.2.4 Model-driven architecture) is a desirable feature for a model because abstraction increases model portability. No single platform is used for the whole QAMT automation model but several platforms for the different model elements. Table 15 presents the platform independence evaluation of each QAMT model element. Although, in the previous section, the stylebase was evaluated as an assisting element for marks, mapping and record, the stylebase is here evaluated only as an individual evaluation element because the stylebase has different platforms from those of marks, mapping and record.

Table 15. Platform independence of the QAMT model.

Element or Actor	Platform independence	
	Independent of	**Dependent on**
Stylebase	• Knowledge base implementation e.g. linked object list, SQL database • Style representation in knowledge base e.g. textual representation, graphical style templates	-
Marks	Mark implementation: • Marking language • Marking mechanism	Modelling language (needs to support components)
Mapping	• Knowledge base implementation e.g. linked object list, SQL database • Rule representation in knowledge base e.g. natural language, mapping language • Transformation tool	-
Record	Record implementation: • Recording language • Recording mechanism Modelling tool	-

4.2.3 Level of automation

Automation models for each of the three main activities of the QAMT model are provided in Figures 8–10, whereas the rest of the activities are left on a high abstraction level, i.e. the automation of those activities is not considered. Table 16 describes the level of automation in the three main activities. The automation level of the model is not consistent across the different model activities. This may be due to the reasons discussed in the following. Updating an architectural model is the most trivial activity in the transformation and therefore the easiest to automate, whereas identifying source and target are the most difficult activities to automate. Trivial activities do not require complex reasoning made by humans. The difficult activities can be automated only if there is an explicit mapping between quality attributes, requirements, and software structures. This is, however, difficult or even impossible to achieve due to the current state-of-the-art (See page 37).

Table 16. Level of automation in the QAMT model.

Activity	**Sub activity**	**Level of model automation**		
		Manual	**Semi-automatic**	**Automatic**
Identify source	Study the model		√	
	Evaluate the model	√		
	Select source		√	
Identify target	Search target candidates		√	
	Evaluate candidate	√		
	Select target		√	
Trans-formation	Receive source and target			√
	Update architectural model			√

4.2.4 Maturity

The maturity of the model is validated in practice. QAMT model activities are validated in five individual cases. See case descriptions in Table 1 on page 22. Further, Table 17 provides an overview on the cases by pointing out *which* activities were validated in each case. Below, I summarize *what* was done in each activity.

Case C1. A new extensibility requirement required an architecture transformation from layered to blackboard. Model activities were first done manually and then automated by developing a platform specific implementation model and a tool prototype (Merilinna 2005).

Cases C2 and C3. Case C2 was developed with the quality properties modifiability, integrability and portability. Case C3 showed a totally different functionality but the same quality properties as C2 and, therefore, no trigger for QAMT was observed. The transformation of the architecture model was not relevant although the activities of identifying source and target were done.

Case C4. The first sketch of the pilot was developed with the emphasis on basic functionality and with no specified quality concerns. The architecture was transformed twice to improve quality properties. On the server side, first, the architecture was transformed in order to improve real-time performance. Second, in order to improve modifiability, the separation of concerns in the server architecture was improved.

Table 17. Validation of QAMT model activities in cases.

Case ID	**Case description**	**Model activities**		
		Identify source	**Identify target**	**Trans-formation**
C1	Complete QAMT automation trial with a laboratory case	Manual, Semi	manual, semi	manual, automated
C2, C3	Trial to transform architecture from case C2 to case C3. No transformation trigger observed, variability only in functionality	Manual	manual	-
C4	Trial to develop a pilot and transform the architecture twice.	Manual	manual	manual
C5	Trial to evaluate architecture, suggest appropriate transformations and estimate the effects of transformations	Manual	manual	simulated

Case C5. Existing and working products were already on the market. Transformation trigger was defined as the emergence of a new hardware and the wish that the architecture should support both the existing and new hardware. These requirements would require new quality properties from the product architecture: portability, modifiability and extensibility. The aim of the trial was to identify source, i.e. parts of the architecture that would require changes, to identify target, i.e. to suggest appropriate solutions, and third, to simulate transformation, i.e. to estimate which components would be affected and what kind of changes were required, i.e. to estimate the effect of each scenario on the architecture.

4.2.5 Conformance to standards

A short description of the appropriate standards related to quality-driven software architecture transformation and a discussion on the conformance of QAMT to the standards are provided below.

The UML 2.0 Superstructure Specification (OMG 2005b) is a modelling language description providing the syntax and semantics of the language and, further, a few examples on how to draw diagrams with the language. UML 2.0 is utilized in the QAMT model in two ways. First, the language is utilized in creating the architectural models for transformation. Architectural models for transformation are provided by the QADA method, which guides the use of UML2 for describing software architecture (Immonen & Niskanen 2005, Merilinna 2005). The UML2 language constructs that are appropriate for describing software architectures have their origin in the ROOM (Real-time object oriented modelling) method (Selic et. al 1994), which later became part of the UML2 standard. The constructs of the ROOM method have already been adopted in the earlier versions of QADA (Paper III, Matinlassi et al. 2002).

Second, the UML2 language is utilized to describe the diagrams of the QAMT automation model. Activity diagrams and a collaboration diagram were utilized especially in the section 3 of this thesis.

The ISO/IEC 9126-1 Quality model (ISO-9126-1 2001) is a quality model for software products (see also Section 2.1.1 Quality properties). It provides six quality characteristics and sub characteristics for an intermediate product, i.e. product in development, and four quality characteristics for a product in use. The characteristics claim to provide a consistent terminology for software product quality. However, the research forum is a step ahead from the standardization forum. Furthermore, standardization is a heavy process while the research forum continuously produces new information on the subject. Therefore, the terminology standardized in 2001 was probably already outdated when published. The QAMT model uses the current quality terminology applied in the research field (described in Section 2.1.1). The major difference between the standard and current research is that the former considers functionality as a quality attribute whereas the latter makes a clear distinction between the functional and non-functional properties of software.

Model Driven Architecture (OMG 2003a) is not really a standard but an initiative, which proposes to define a set of non-proprietary standards that will specify interoperable technologies with which to realize model-driven development with automated transformations (see Section 2.2.4 Model Driven

Architecture). The terminology and ideology of the initiative is applied throughout the QAMT model.

The Query/Views/Transformation mapping language (OMG 2002) is not yet a standard either (see the current state of standarization on page 46). The emerging mapping language standard will define (1) a language for making *queries* for MOF models, (2) a language that enables the creation of *views* for a model and (3) a language for defining *transformations*. The QVT mapping language is not utilized in QAMT because the standard is still maturing and, furthermore, because the QAMT mapping is not yet complete enough (see Section 4.2.1 Completeness) to be translated into a dedicated mapping language.

The IEEE Recommended practice for architectural descriptions (IEEE-1471 2000) is a standard describing the terminology and interdependencies between the terms concerning architectural descriptions. The standard also recommends basic principles for architectural documentation including viewpoints, views and rationale. The terminology of the standard is utilized throughout the QAMT model. The models provided as an input for quality-driven architecture transformation also follow the recommended practice for architectural documentation (conformance of architectural models to IEEE standard 1471-2000 is discussed in Paper III).

4.3 Evaluation summary

The QAMT model defines model activities and actors that execute the activities. The actors correspond to the transformation elements defined in (OMG 2003a). The evaluation of the QAMT model was performed from the two points of view provided by transformation actors and model activities. The evaluation of model elements and actors considered the *completeness* and *platform independence* of elements/actors, whereas the evaluation of model activities considered the *automation level* and *maturity* of model activities. Furthermore, the conformance of the QAMT model to the most important standards in the area was evaluated.

This summary provides the evaluation results in tables (Table 18, 19 and 20) and also discusses the internal consistency of the model, i.e. to which extent the different parts of the model show a uniform level of completeness, platform independence, automation and maturity.

Table 18. Summary of QAMT element evaluation.

Criterion / Element	Completeness	Platform independence
Stylebase	The stylebase includes valid parameters in order to support: model marking, constraining and defining mapping rules, generating transformation record and end-user convenience.	Independent of programming language and modelling tool, dependent on modelling language Independent of knowledge base implementation and style representation in knowledge base
Marks	Marks are supported by associating the following stylebase data to model components: component type, style and role	Independent of marking language and mechanism, dependent on model
Mapping	Mapping is supported in the rulebase. Rules are not complete: • Only admissibility rules defined (with natural language) • A technique for defining feasibility rules • Standard mapping language not utilized	Independent of knowledge base implementation, rule representation in knowledge base and transformation tool
Transformation record	Transformation record supported in the rule definition technique	Independent of recording language and mechanism, modelling tool

Table 19. Summary of QAMT activity evaluation.

Activity	**Criterion**	**Level of automation**			**Validation in cases C1–C5**				
		Manual	Semi	Auto	1	2	3	4	5
Identify source	Study the model		√		√	√	√	√	√
	Evaluate the model	√							
	Select source		√						
Identify target	Search target candidates		√		√	√	√	√	√
	Evaluate candidate	√							
	Select target		√						
Transfor-mation	Receive source and target			√	√	-	-	√	√
	Update architectural model			√					

Table 20. QAMT conformance to standards.

Standard	**How applied?**
UML2	Superstructure specification applied – as an architecture modelling language – as the language of the QAMT model
MDA	Terminology and ideology applied in – transformation specification and in – architecture modelling
IEEE-1471	Terminology and recommendations applied in – the structure of architectural descriptions

The internal consistency of each criterion is summarized below. Summarizing the consistency of completeness, I may draw the conclusion that the marks, record and the stylebase are the most complete elements, whereas mapping rules suffer from a lack of completeness (Figure 14).

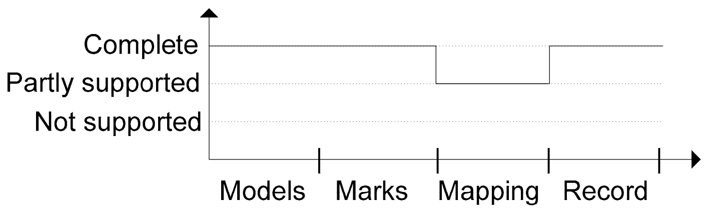


Figure 14. Internal consistency of completeness in the QAMT automation model.

The abstraction level of the QAMT automation model is relatively high, because it is only dependent on the modelling language, and dependence on the modelling language is already a pre-requirement in MDA. The internal consistency of abstraction throughout the QAMT automation model is fairly uniform (Figure 15). Only the model with marks is dependent on modelling language, whereas stylebase, rulebase and record are independent of any platform.

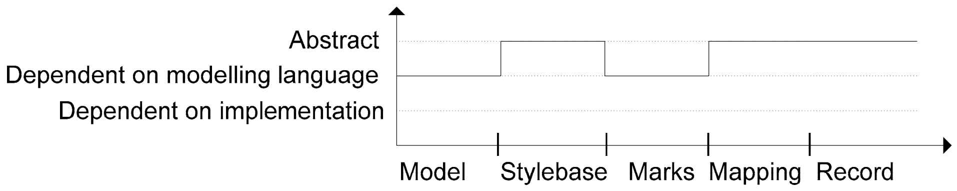


Figure 15. Internal consistency of platform independence in the QAMT automation model.

The automation level of the model is not consistent across the model activities (Figure 16). Updating the architectural model is the most trivial activity in the transformation and therefore it is the easiest activity to automate, whereas identifying source and target are the most difficult activities to automate.

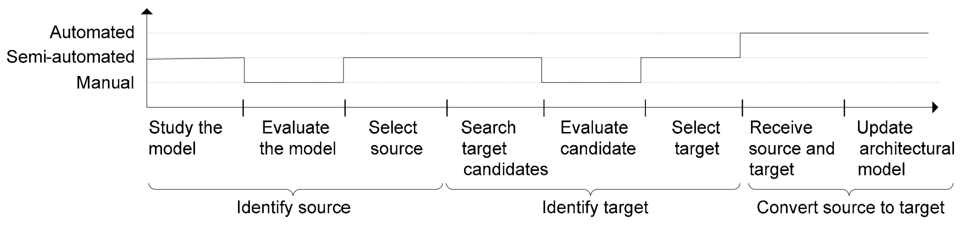


Figure 16. Internal consistency of level of automation in the QAMT model.

The ‘Convert source to target’ activity is less mature than the other activities included in the QAMT model (Figure 17). Figure 17 illustrates how many times each activity was validated in the different cases. “Identify source” and “identify target” were done five times, whereas “convert source to target” was validated only three times.

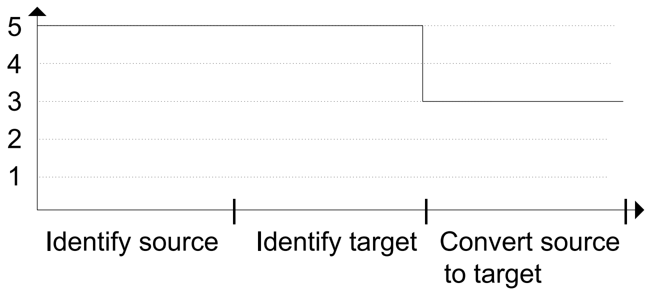


Figure 17. Internal consistency of validation in the QAMT model.

The goals of the model were to (1) describe transformation as completely as possible, (2) provide support towards automation, (3) stay independent of implementation technologies, (4) be mature and validated and (5) conform to standards. The model was evaluated against these criteria. As a result, it was concluded that the QAMT model describes transformation quite completely, while only mapping suffers lack of explicit specification. QAMT does not totally automate all the model activities but it reduces the need for human intervention while identifying the source and target and completely eliminates the need for human intervention in the transformation activity, except for some individual transformation cases that still need human interaction in transformation. The QAMT model succeeds well in platform independence and is validated in five individual cases, although not consistently, i.e. different cases cover different parts of the model. Finally, the QAMT model promotes understandability by following, e.g., the terminology and specification structure defined in the most important standards applicable in the area.

5. Conclusions

This section concludes the dissertation by presenting the summary of the results, the limitations of results, and outlining the future research. The summary of the results draws a conclusion to the research question and summarizes how the research question was answered in the papers and in the dissertation summary. The limitations of the results discuss the validity and applicability of the results. Future research section points out both the incomplete and the most robust areas of the dissertation and draws out a future research plan to complement and continue the work.

5.1 Summary of the results

Quality-driven software architecture model transformation is about making changes to an architecture model according to changing or varying quality properties. The automation of quality-driven software architecture model transformation will reduce human involvement in the modelling process and therefore decrease software development costs. On the other hand, although automation does not eliminate the errors made in software development, it increases the probability of higher quality for the product. The research question studied in this dissertation was the following.

How to make automation of quality-driven software architecture model transformation (QAMT) possible?

The answer to the research question is presented in seven papers and in the dissertation summary. Automation is made possible by defining and unifying the knowledge needed in quality-driven software architecture model transformation in form of a *transformation* specification. The transformation specification includes a model, marks, a mapping and a transformation record, wherein a model definition provides the foundation stone for the specification while the marks, the mapping and the record complement the specification. The model used in this transformation specification is a *quality-driven architecture model* and it is covered in the dissertation as follows.

The concept of architecture *quality* is discussed in Papers V and VI by presenting the terminology and a quality model for software architecture. Section 2.1 updates and unifies the terminology presented in earlier papers.

Papers II and III concentrate on defining how an *architectural model* is described using architectural viewpoints. The state-of-the-art of architecture modelling is updated and refined towards model driven architecture in Section 2.2. The definition of architectural model descriptions required studying and comparing the existing methods for architecture design (Paper I & III). The state-of-the-art of transformation is presented in Section 2.3 and a definition of complementing transformation elements (marking, mapping, record) is presented in Section 3 in this dissertation summary. Section 3.3 focuses on defining the actors that *make the automation possible*.

The experience of designing software *architecture* gained in the different cases is presented in Papers IV, V, VI and VII. In addition to the empirical validation through the cases, the model is validated through the self-evaluation presented in the dissertation summary, Section 4.

5.2 Limitations of the results

The research results have two specific limitations. The first limitation has to do with the extent to which the results can be generalized beyond the cases studied. The number of cases is too limited for broad generalizations. However, the intention was not to produce a general model for automating architectural transformations but rather to create a starting point for model transformations in software architecture. Partly due to the limited number of cases involved in this research, the resulting model does not explicitly define the processes used by a software engineer during architectural transformation or automate all the complex reasoning done during the process. However, exact fidelity to real world phenomena was not the goal of the QAMT model, but rather to provide a model that would be accurate enough for approaching the automation of model activities. Furthermore, a model is generally a simplified representation of the real phenomenon. Accordingly, the aim was to simplify the design process in order to make automation possible. To facilitate automation, a simplified model is created of the factual process, the model is then split up into smaller functional pieces and these pieces are automated one by one. The model does not make any attempt to provide the ‘right’ way to automate transformation either. This is partly due to the fact that the ways (or approaches) of automating a transformation are manifold (see Section 2.3). The results aim to provide an outcome that is as platform independent as possible, thereby improving the applicability of the model.

The second limitation concerns the applicability of the model in product families. The adoption of an automated QAMT in product family architecture, however, sets certain requirements for the PFA implementation. Product family architectures that are mature and stable enough will be able to derive the most benefit from adopting automated QAMT. Referring, e.g., to the maturity levels defined in (van der Linden et al. 2004), to make the most of automating QAMT, the product family architecture shall be at level 4 or 5, i.e. variant products or self-configurable products. The three lowest levels of product family maturity, namely independent components, standardized infrastructure and software platform, are considered too immature as product family engineering approaches for adopting automated QAMT. In practice, the fourth product family maturity level means that a PFA may be implemented either with (1) a software architecture that enables systematic product derivation according to the given PFA or (2) with configurable features or a component base where architecture is integrated into the platform and also into the common component base (Niemelä 2005). In the most mature case of PFA (the fifth level), PFA is implemented using a configurable product family base, wherein the product family members are automatically generated according to the architecture.

5.3 Future research

This research has introduced an automation model for quality-driven software architecture model transformation, which is employed as a means to move towards automating quality-driven software architecture model transformation. Here, I will draw out a future research plan to complement and continue the work done in this dissertation.

The evaluation of the automation model revealed that the mapping specification was the most incomplete part of the model. Therefore, the future research will start by completing the mapping part of the transformation specification in natural language and then translating the mapping into a dedicated mapping language. This will make mapping compatible with any modelling tool supporting the mapping language. This will serve the purpose of increasing the level of automation in this area.

However, in order to remarkably increase the level of automation, more explicit knowledge on the qualities promoted by styles and patterns is needed. Today, this knowledge is context sensitive and also dependent on the experience and skills of the architect. Further, while each style has a prime purpose for which it can be applied, styles may also be adapted and applied for other than the prime purposes. The quality attributes supported by a style also depend on such factors as system size and domain (e.g. distributed system), as presented in Paper VI. Further, experienced architects have advanced knowledge on applying particular styles in different contexts, which allows them even to easily figure out new styles. The skills, background, opinions and other properties of the architect affect the content and constitution of the quality attributes used by the architect and these factors may be reflected in the way styles are applied to support the quality attributes. This information is required for transformation, and, furthermore, it is only possible to perform transformation if it relies on a set of uniform quality attribute definitions.

In addition to the topics above, the stylebase shall be also developed into a more advanced knowledge repository and the implementation of the stylebase will have to be experimented with new ideas. Further, the stylebase will be integrated with quality properties, thereby mapping the road towards automated transformation from quality requirements to architectural styles. Thus, what is also needed is support for representing the evolution qualities in architectural models. For now, UML profiles are used for describing the execution quality properties in a model (See Section 2.1.3, example quality profiles for fault tolerance, time, schedulability, performance, reliability and availability). Similar kind of support, perhaps in form of quality profiles, will also be needed for utilizing evolution qualities in automating architecture model transformations.

So far, the research effort on model driven architecture has been focusing on automating vertical transformations, such as code generation, and on defining standard mapping languages from models to models and from models to text.

The work in this dissertation initiates the automation of horizontal model transformations and the work is still in progress. The future research topics suggested above will accumulate the knowledge on the subject and also derive fresh topics to experiment with new ideas.

6. Introduction to the papers

This section gives an overview on the original papers constituting the basis of this dissertation. Table 21 presents basic information of the papers and illustrates their main considerations. A more detailed presentation of the contributions of the papers is given in Table 3 (Summary of publications related to dissertation research) and Table 4 (Contributions of Papers I–VII and DS for dissertation research) on pages 26–27. The following sections discuss the considerations and observations of the papers.

Table 21. Original papers and their main considerations.

Original paper	**Published in, forum**	**Considers mainly**		
		State of the art	**Method development**	**Cases**
Paper I	2004, ICSE	√		
Paper II	2002, ICSSEA		√	
Paper III	2004, Journal of Systems and Software, Vol. 69		√	
Paper IV	2002, Profes			√
Paper V	2003, Euromicro			√
Paper VI	2003, Annual Review of Communications, Vol. 56			√
Paper VII	2002, MUM			√

6.1 State of the art

6.1.1 Paper I: Design method comparison

Paper I presents a study of comparing product family architecture methods by developing an evaluation framework for comparing the design methods and introducing and comparing five methods that are known to answer the needs of software product families: COPA, FAST, FORM, KobrA and QADA.

The main consideration of the paper is that the methods studied show distinguishable ideologies making the methods not overlapping or competing with each others. Paper I serves both for revealing state-of-the-art of product family architecture design methods and also as a comparative analysis of the QADA methodology and other similar methods.

6.2 Method development

6.2.1 Paper II: Introducing the design method

Paper II introduces the first release of the quality-driven architecture design method with two abstraction levels: conceptual architecture design and concrete architecture design. The architectural descriptions at both abstraction levels are defined from three viewpoints: structural, behaviour and deployment. The paper also shortly introduces case C1 and discusses the case experiences.

6.2.2 Paper III: Refining the design method

Paper III puts forward the following points in the quality-driven architecture design method. First, a development view is provided, and second, the viewpoints are described according to standard viewpoint description guidelines. The paper also introduces a perspective to the viewpoints needed for developing digital signal processing software and provides a comparison and analysis of the defined viewpoints in two domains. A comparison in Paper III shows that domain and system size are the dominant issues to be considered when architectural viewpoints are being selected.

6.3 Cases

6.3.1 Paper IV: Interactive gaming service

Paper IV introduces the problem overview of case C4 and summarises the initial non-functional requirements of the case: portability, maintainability, integratability, and simplicity. The viewpoints of the QADA methodology are adapted to suit especially the wireless domain and the paper provides some example diagrams for the viewpoints. The discussion section of the paper is concerned with the learning curve of the method in a multinational software development team, while it also focuses on tool support and tool use learning, along with presenting some experiences on how well the viewpoints served the purposes of the stakeholders.

6.3.2 Paper V: Terminal software product family

Paper V introduces case C5, a product family for different kinds of client terminals used for fare collection in public transportation. Further, Paper V introduces a framework for maintainability. The main considerations of the paper are the following:

- Maintainability means different things for different parts of the system with different dimensions, e.g. system, architecture and single component in the architecture
- Not all the 'ilities' are non-functional requirements, e.g. traceability, variability, tailorability and monitorability are techniques for promoting and supporting the achievement of maintainability and its sub-attributes.

6.3.3 Paper VI: Service architectures

Paper VI refines case C4 by introducing the stakeholders of the case and illustrating how non-functional requirements are derived from stakeholders. Further, the non-functional requirements (portability, maintainability, integratability, and simplicity) introduced in Paper IV are mapped to viewpoints introduced in Paper II. Finally, the case C4 architecture is evaluated against the non-functional requirements set in the beginning. Summarizing, the main considerations in the paper are the following:

- The quality properties of software are derived from the stakeholders' needs, while quality accumulates through cooperation with the stakeholders
- Further justification for the necessity of two separate levels of abstraction and the need for multiple viewpoints in architectural representations.

6.3.4 Paper VII: Middleware multimedia services

Paper VII introduces two cases (C2 and C3), which have to do with service platform development for multimedia applications. C2 provides a streaming service, and C3 is concerned with instant messaging and presence services for various types of multimedia applications. Among the key observations presented in the paper are:

- Although having different functional requirements, both platforms conform to similar architectures because of convergent quality requirements: modifiability, integrability and portability.
- The dominant architectural styles in the cases (blackboard and layered styles) achieved the qualities of modifiability, integrability and portability
- In addition to architectural styles, also design level choices affect software quality. Interoperability, simplicity and maintainability, for example, are influenced even at the design level.

References

Abrahamsson, P. 2002. The role of commitment in software process improvement. Oulu: Oulu University Press. 162 p. (Acta Universitatis Ouluensis, Scientiae Rerum Naturalium A386.) ISBN 951-42-6729-X.

van Aken, J. E. 2004. Management research based on the paradigm of the design sciences: The quest for field-tested and grounded technological rules. Journal of Management Studies, Vol. 41, No. 2, pp. 219–246. ISSN (printed): 0022-2380. ISSN (electronic): 1467-6486.

Al-Naeem, T., Gorton, I., Babar, M. A., Rabhi, F. & Benatallah, B. 2005. A quality-driven systematic approach for architecting distributed software applications. Proceedings of the 27th International Conference on Software Engineering, ICSE 2005. St. Louis, Missouri, USA, 15–21 May 2005. New York, NY, USA: ACM Press. Pp. 244–253. ISBN 1-59593-963-2.

America, P., Obbink, H., Muller, J. & van Ommering, R. 2000. Copa: A component-oriented platform architecting method for families of software intensive electronic products. Tutorial in the First Conference on Software Product Line Engineering, SPLC1. Denver, Colorado, August 28–31 2000.

America, P., Obbink, H., van Ommering, R. & van der Linden, F. 2000. Copam: A component-oriented platform architecting method family for product family engineering. In: Donohoe, P. (Ed.). Software product lines, experience and research directions, proceedings of the first software product lines conference, SPLC1. Denver, Colorado, USA, August 28–31 2000. Boston: Kluwer Academic Publishers. (Kluwer international series in engineering and computer science Vol. 576.) Pp. 167–180. ISBN 0-7923-7940-3.

Anastasopoulos, M. & Gacek, C. 2001. Implementing product line variabilities. Symposium on Software Reusability, SSR'01. Toronto, Ontario, Canada, 18–20 May 2001. USA: ACM. (Software Engineering Notes Vol. 26, No. 3.) Pp. 109–117. ISSN 0163-5948.

Andersson, J. & Bosch, J. 2005. Development and use of dynamic product-line architectures. IEE Proceedings – Software, Vol. 152, No. 1, pp. 15–28. ISSN 1462-5970.

Andersson, J. & Johnson, P. 2001. Architectural integration styles for large-scale enterprise software systems. Proceedings of the Fifth IEEE International Enterprise Distributed Object Computing Conference, EDOC'01. Seattle, WA, USA, 4–7 September 2001. Los Alamitos, California: IEEE Comput. Soc. Pp. 224–236. ISBN 0-7695-1345-X.

Ardis, M., Daley, N., Hoffman, D., Siy, H. & Weiss, D. 2000. Software product lines: A case study. Software Practice and Experience, Vol. 30, No. 7, pp. 825–847. ISSN 0038-0644.

Aßmann, U. (ed.). 2004. Proceedings of model-driven architecture: Foundations and applications, Linköping: Linköping University. 253 p.
http://www.ida.liu.se/~henla/mdafa2004/proceedings.pdf.

Bachmann, F. & Bass, L. 2001. Managing variability in software architectures. Symposium on Software Reusability, SSR'01. Toronto, Ontario, Canada, 18–20 May 2001. USA: ACM. (Software Engineering Notes Vol. 26, No. 3.) Pp. 126–132. ISSN 0163-5948.

Bass, L., Clements, P. & Kazman, R. 1998. Software architecture in practice. Reading, Massachusetts: Addison-Wesley. 452 p. ISBN 0-201-19930-0.

Becker, M., Geyer, L., Gilbert, A. & Becker, K. 2002. Comprehensive variability modelling to facilitate efficient variability treatment. Proceedings of the 4th International Workshop in Software Product-Family Engineering, Bilbao, Spain, 3–5 October 2001. Berlin, Germany: Springer-Verlag. (Lecture Notes in Computer Science 2290.) Pp. 294–303. ISBN 3-540-43659-6.

Bettin, J. 2005. Model-driven software development. In: Frankel, D. & Parodi, J. MDA journal: Model driven architecture straight from the masters. Tampa, FL, USA: Meghan-Kiffer Press. ISBN 0-92965-225-8. Available online:
http://www.bptrends.com/publicationfiles/04%2D04%20COL%20MDSD%20Frankel%20%2D%20Bettin%20%2D%20Cook%2Epdf.

Bézivin, J. 2004. On the basic principles of model engineering. In: Gérard, S., Champeau, J. & Babau, J.-P. (eds.). Proceedings of the second summer school, MDA for Embedded Systems. Brest, Brittany, France, 6–10 September 2004. France: ENSIETA. Part I. Pp. 1–47.

Birbilis, G., Koutlis, M., Kyrimis, K., Tsironis, G. & Vasiliou, G. 2000. E-slate: A software architectural style for end-user programming. Proceedings of International Conference on Software Engineering. Limerick, Ireland, 4–11 June 2000. Los Alamitos, California, USA: IEEE Computer Society. Pp. 684–687. ISSN 0270-5257.

Booch, G., Brown, A., Iyengar, S., Rumbaugh, J. & Selic, B. 2005. An MDA manifesto. Frankel, D. & Parodi, J., MDA journal: Model driven architecture straight from the masters. Tampa, FL, USA: Meghan-Kiffer Press. ISBN 0-92965-225-8. Available online: http://www.bptrends.com/publicationfiles/05-04%20COL%20IBM%20Manifesto%20-%20Frankel%20-3.pdf.

Born, M., Schieferdecker, I., Kath, O. & Hirai, C. 2005. Combining system development and system test in a model-centric approach. In: Guelfi, N. (ed.). Revised selected papers from the first international workshop in rapid integration of software engineering techniques, RISE2004. Luxembourg-Kirchberg, Luxembourg, 26 November 2004. Berlin, Germany: Springer-Verlag. (Lecture Notes in Computer Science Vol. 3475) Pp. 132–143. ISBN 3-540-25812-4.

Bosch, J. 2000. Design and use of software architectures: Adopting and evolving a product-line approach. 1st edition. Harlow: Addison-Wesley. 368 p. ISBN 0201674947.

Bosch, J. Florijn, G., Greefhorst, D., Kuusela, J., Obbink, H.J. & Pohl, K. 2002. Variability issues in software product lines. Proceedings of the 4th International Workshop, PFE 2001. Bilbao, Spain, 3–5 October 2001. Revised Papers. (Lecture Notes in Computer Science 2290.) Berlin, Germany: Springer-Verlag. Pp. 13–21. ISBN 3 540 43659 6.

Bosch, J. & Molin, P. 1999. Software architecture design: Evaluation and transformation. Proceedings of IEEE Conference and Workshop on Engineering of Computer-Based Systems. Nashville, TN, USA, 7–12 March 1999. Los Alamitos, CA, USA: IEEE Computer Society. Pp. 4–10. ISBN 0769500285.

Bratthall, L. & Runeson, P. 1999. A taxonomy of orthogonal properties of software architecture. Proceedings of the second Nordic software architecture workshop, NOSA'99. University of Karskrona, Ronneby, Sweden, 12–13 August 1999.

Brown, A. 2004. An introduction to model driven architecture – part 1: MDA and today's systems. IBM. Rational Edge. Electronic magazine. [Referenced 4.1.2006]. URL www-106.ibm.com/developerworks/rational/library/3100.html.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. & Stal, M. 1996. Pattern-oriented software architecture – a system of patterns. 1st edition. Chichester, New York: John Wiley & Sons. 456 p. ISBN 0471958697.

Cheng, S.-W., Garlan, D., Schmerl, B., Sousa, J. P., Spitznagel, B. & Steenkiste, P. 2002. Using architectural style as a basis for system self-repair. Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture, WICSA3. Montreal, Quebeck, Canada, 25–30 August 2002. Norwell, MA, USA: Kluwer Academic Publishers. Pp. 45–59. ISBN 1 4020 7176 0.

Christoph, A. 2004. Describing horizontal model transformations with graph rewriting rules. Proceedings of Model driven architecture: European MDA workshops: Foundations and applications, MDAFA 2003 and MDAFA 2004. Twente, the Netherlands, June 26–27 2003 and Linkoping, Sweden, June 10–11 2004. Revised selected papers. (Lecture Notes in Computer Science 3599.) Heidelberg, Germany: Springer-Verlag. Pp. 93–107. ISSN 0302-9743.

Chung, L., Nixon, B., Yu, E. & Mylopoulos, J. 2000. Non-functional requirements in software engineering. Boston, Dordrecht: Kluwer Academic Publishers. 439 p. ISBN 0-7923-8666-3.

Czarnecky, K. & Helsen, S. 2003. Classification of model transformation approaches. Workshop on generative techniques in the context of model-driven architecture in ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA’03. Anaheim, California, USA, 26–30 October 2003.

Dobrica, L. & Niemelä, E. 2000. Attribute-based product-line architecture development for embedded systems. Proceedings of the 3rd Australasian workshop on software and systems architectures, AWSA’2000. Sydney, Australia, 19–20 November 2000. Pp. 76–88.

Dobrica, L. & Niemelä, E. 2002. A survey on software architecture analysis methods. IEEE Transactions on Software Engineering, Vol. 28, No. 7, pp. 638–653. ISSN 0098-5589.

Dueñas, J. C., de Oliveira, W. & de la Puente, J. 1998. A software architecture evaluation model. Proceedings of the second international ESPRIT ARES workshop on development and evolution of software architecture for product families. Las Palmas de Gran Canaria, Spain, 26–27 February 1998. Berlin, Germany: Springer-Verlag. Pp. 148–157. ISBN 3-540-64916-6.

Fenkam, P., Gall, H., Jazayeri, M. & Kruegel, C. 2002. DPS: An architectural style for development of secure software. Proceedings of International conference on infrastructure security, InfraSec 2002. Bristol, UK, 1–3 October 2002. Berlin, Germany: Springer-Verlag. (Lecture Notes in Computer Science Vol. 2437.) Pp. 180–198. ISBN 3 540 44309 6.

Foley, J. D. & van Dam, A. 1982. Fundamentals of interactive computer graphics. Reading: Addison-Wesley. 664 p. ISBN 0-201-14468-9.

Frankel, D. 2003. Model driven architecture, applying MDA to enterprise computing. New York: Wiley. 328 p. ISBN 0-471-31921-1.

Frankel, D. 2005. Eclipse and MDA. David Frankel Consulting, Business Process Trends. Web column. [Referenced 4.1.2006].
URL http://www.bptrends.com/

Gall, H., Jazayeri, R., Klosch, R. & Trausmuth, G. 1997. The architectural style of component programming. Proceedings of the 21st Annual International Computer Software and Applications Conference, COMPSAC'97. Washington, DC, USA, 13–15 August 1997. Los Alamitos, CA: IEEE Computer Society. Pp. 18–25. ISBN 0 8186 8105 5.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. 1994. Design patterns: Elements of reusable object-oriented software. Addison Wesley. 416 p. ISBN 0-201-63361-2.

Gardner, T., Griffin, C., Koehler, J. & Hauser, R. 2003. A review of OMG MOF 2.0 QVT submissions and recommendations towards the final standard. In: Evans, A., Sammut, P. & Willans, J. S. (eds.). Proceedings of the 1st international workshop on metamodeling for MDA. Kings Manor, York, England, 24–25 November 2003. Pp. 178–197.
URL http://www.cs.york.ac.uk/metamodel4mda/onlineProceedingsFinal.pdf.

Grønmo, R., Aagedal, J., Solberg, A., Belaunde, M., Rosentrhan, P., Faugere, M., Ritter, T. & Born, M. 2005. Evaluation of the QVT Merge language proposal. White paper, 31st of March. [Referenced 4.1.2006]. 85 p. URL http://www.modelware-ist.org/public_area/publications/white_papers/QVT-Eval-OMGReport.pdf.

Grunske, L. 2003. Automated software architecture evolution with hypergraph transformation. Proceedings of the Seventh IASTED International Conference on Software Engineering and Applications. Marina del Rey, CA, USA, 3–5 November 2003. Galgery- Alberta, Canada: International Association of Science and Technology for Development. Pp. 613–620. ISBN 0889863946.

van Gurp, J., Bosch, J. & Svahnberg, M. 2001. On the notion of variability in software product lines. Proceedings of Working IEEE/IFIP Conference on Software Architecture, WICSA 2001. Amsterdam, Netherlands, 28–31 August 2001. Los Alamitos, CA, USA: IEEE Computer Society. Pp. 45–54. ISBN 0 7695 1360 3.

Günther, J. & Steenbergen, C. 2004. Application of MDA for the development of the DATOS billing and customer care system. In: van Sinderen, M. J. & Ferreira Pires, L. (Eds.). Proceedings of the 1st European workshop on model-driven architecture with emphasis on industrial applications, MDA-IA 2004. University of Twente, Enschede, The Netherlands, 17–18 March 2004.

Netherlands: University of Twente. (CTIT Technical Report TR-CTIT-04-12.) Pp. 53–62. ISSN 1381-3625.

Hofmeister, C., Nord, R. & Soni, D. 2000. Applied software architecture. Reading, MA: Addison-Wesley. 397 p. ISBN 0-201-32571-3.

IEEE-1471. 2000. IEEE recommended practice for architectural descriptions of software-intensive systems. New York: IEEE. 23 p.

Immonen, A. 2006. A method for predicting reliability and availability at the architectural level. In: Käkölä, T. & Dueñas, J. C. (Eds.). Software Product Lines: Research Issues in Engineering and Management. Berlin, Heidelberg, New York: Springer. Pp. 373–422. ISBN-10 3-540-33252-9, ISBN-13 978-3-540-33252-7.

Immonen, A. & Niskanen, A. 2005. A tool for reliability and availability prediction. Proceedings of the 31st Euromicro conference on Software Engineering and Advanced Applications, Euromicro 2005. Porto, Portugal, 30 August – 3 September 2005. Los Alamitos, CA, USA: IEEE Computer Society. Pp. 416–423.

ISO-9126-1. 2001. Software engineering – product quality – part 1: Quality model. ISO/IEC. 25 p.

Jaaksi, A., Aalto, J.-M., Aalto, A. & Vättö, K. 1999. Tried & true object development: Industry-proven approaches with UML. Cambridge Univ.: Cambridge University Press. 315 p. ISBN 0-521-64530-1.

Järvinen, P. 2001. Improving the quality of drawings. Computers and Networks in the Age of Globalization. Proceedings of the IFIP TC9 world conference on human choice and computers. Geneva, Switzerland, 25–28 August 1998. Norwell, MA, USA: Kluwer Academic Publishers. Pp. 245–259. ISBN 0 7923 7253 0.

Järvinen, P. 2004. On research methods. New edition. Tampere, Finland: Opinpajan kirja, Tampereen yliopistopaino Oy. 204 p. ISBN 952-99233-1-7.

Kang, K. C., Cohen, S., Hess, J., Novak, W. & Peterson, A. 1990. Feature-oriented domain analysis (FODA) Feasibility study. Pittsburgh, PA, USA: Software Engineering Institute. 147 p. (SEI Technical Reports CMU/SEI-90-TR-21.)

Karhinen, A., Kuusela, J. & Tallgren, T. 1997. An architectural style decoupling coordination, computation and data. Proceedings of the third IEEE international Conference on Engineering of Complex Computer Systems. Como, Italy, 8–12 September 1997. Los Alamitos, CA, USA: IEEE Computer Society. Pp. 60–68. ISBN 0 8186 8126 8.

Keshav, R. & Gamble, R. 1998. Towards a taxonomy of architecture integration strategies. In: Magee, J. & Perry, D. (Eds.). Proceedings of the third international software architecture workshop, ISAW-3. Orlando, FL, USA, 1–5 November 1998. New York, NY, USA: ACM. (Foundations of Software Engineering.) Pp. 89–92. ISBN 1-58113-081-3.

Klein, M. & Kazman, R. 1999. Attribute-based architectural styles. Pittsburgh, PA, USA: Software Engineering Institute. 74 p. (SEI Technical reports CMU/SEI-99-TR-022.)

Kobryn, C. 2004. UML 3.0 and the future of modeling. Software and Systems Modeling, Vol. 3, No. 1, pp. 4–8. ISSN 1619-1366 (Paper) 1619-1374 (Online).

Kronlöf, K. 1993. Method integration: Concepts and case studies. Chichester: John Wiley & Sons. 402 p. ISBN 0-471-93555-7.

Kruchten, P. 1995. The 4+1 view model of architecture. IEEE Software, Vol. 12, No. 6, pp. 42–50. ISSN 0740-7459.

Krueger, C. 2004. Introduction to Software product lines. Web page. [Referenced 14.10.2004]. URL http://www.softwareproductlines.com.

Levy, N. & Losavio, F. 1999. Analyzing and comparing architectural styles. Proceedings of XIX international conference of the Chilean computer science society, SCCC'99. Talca, Chile, 11–13 November 1999. Los Alamitos, CA, USA: IEEE Computer Society. Pp. 87–95. ISBN 0 7695 0296 2.

van der Linden, F., Bosch, J., Kamsties, E., Känsälä, K. & Obbink, H. 2004. Software product family evaluation. In: Nord, R. Proceedings of the Third International Software Product Line Conference, SPLC 2004. Boston, MA, USA, 30 August – 2 September 2004. Berlin, Heidelberg: Springer. Pp. 110–129. ISBN 3-540-22918-3.

March, S. T. & Smith, G. F. 1995. Design and natural science research on information technology. Decision Support Systems, Vol. 15, No. 4, pp. 251–266. ISSN 0167-9236.

Matinlassi, M. 2002 Evaluation of Product line architecture design methods. Proceedings of young researchers' workshop in the seventh international conference on software reuse, ICSR7. Austin, Texas, USA, 15–19 April 2002. Web proceedings. [Referenced 4.1.2006]. URL http://www.info.uni-karlsruhe.de/~heuzer/ICSR-YRW2002/papers/MariMatinlassi_Evaluation_of_Product_Line_Architecture_DesignMethods.pdf.

Matinlassi, M. 2004. Evaluating the portability and maintainability of software product family architecture: Terminal software case study. In: Magee, J., Szyperski, C. & Bosch, J. (Eds.). Proceedings of the 4th IEEE/IFIP conference on software architecture, WICSA 2004. Oslo, Norway, 12–15 June 2004. Los Alamitos, CA, USA: IEEE Computer Society. Pp. 295–298. ISBN 0-7695-2172-X.

Matinlassi, M. 2005. Quality-driven software architecture model transformation. Proceedings of the fifth working IEEE/IFIP conference on software architecture, WICSA 2005. Pittsburgh, PA, USA, 6–9 November 2005. Pp. 199–200. ISBN-13: 978-0-7695-2548-8. ISBN-10: 0-7695-2548-2.

Matinlassi, M. & Kalaoja, J. 2002. Requirements for Service Architecture Modeling. Proceedings of the Workshop of Software modeling engineering of UML2002, WISME2002. Dresden, Germany, 30 September – 4 October 2002. Web proceedings. [Referenced 4.1.2006].
URL http://www.metamodel.com/wisme-2002/papers/matinlassi.pdf.

Matinlassi, M. & Niemelä, E. 2002. Designing high quality architectures. Proceedings of the workshop on software quality in ICSE 2002. Orlando, USA, 25 May 2002. 4 p.

Matinlassi, M., Niemelä, E. & Dobrica, L. 2002. Quality-driven architecture design and quality analysis method, a revolutionary initiation approach to a product line architecture. Espoo: VTT Electronics. 129 p. + app. 10 p. (VTT Publications 456). ISBN 951-38-5967-3; 951-38-5968-1. http://virtual.vtt.fi/inf/pdf/publications/2002/P456.pdf.

Matinlassi, M., Pantsar-Syväniemi, S. & Niemelä, E. 2004. Towards service-oriented development in base station modules. In: Trappl, R. (Ed.). Cybernetics and Systems 2003, Proceedings of the 17th European meeting on cybernetics and system research. Vienna, Austria, 13–16 April 2004. Austria: Austrian Society for Cybernetic Studies. Pp. 440–444. ISBN 3-85206-169-5.

Medvidovic, N., Oreizy, P., Robbins, J. E. & Taylor, R. N. 1996. Using object-oriented typing to support architectural design in the C2 style. Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, SIGSOFT '96. San Francisco, CA, USA, 16–18 Oct. 1996. New York, NY, USA: ACM. Pp. 24–32. ISBN 0 89791 797 9.

Merilinna, J. 2005. A tool for quality-driven architecture model transformation. Espoo: VTT Electronics. 106 p. + app. 7 p. (VTT Publications 561). ISBN 951-38-6439-1;951-38-6440-5. http://virtual.vtt.fi/inf/pdf/publications/2005/P561.pdf.

Merilinna, J. & Matinlassi, M. 2004. Evaluation of UML tools for model-driven architecture. Proceedings of the 11th Nordic workshop on programming and software development tools and techniques, NWPER’2004. Turku, Finland, 17–19 August 2004. Turku: Åbo Akademi University. Pp. 155–165. ISBN 952-12-1385-X.

Monroe, R. T., Kompanek, A., Melton, R. & Garlan, D. 1997. Architectural styles, design patterns, and objects. IEEE Software, Vol. 14, No. 1, pp. 43–52. ISSN 0740-7459.

Nechypurenko, A., Lu, T., Deng, G., Schmidt, D. & Gokhale, A. 2004. Applying MDA and component middleware to large-scale distributed systems: A case study. In: van Sinderen, M. J. & Ferreira Pires, L. (Eds.). Proceedings of the 1st European workshop on model-driven architecture with emphasis on industrial applications. University of Twente, Enschede, The Netherlands, 17–18 March

2004. Netherlands: University of Twente. (CTIT Technical Report TR-CTIT-04-12.) Pp. 1–10. ISSN 1381-3625.

Neelamkavil, F. 1987. Computer simulation and modeling. 1 edition. John Wiley & Sons Inc. 324 p. ISBN 0471911291.

Niemelä, E. 1999. A component framework of a distributed control systems family. Espoo: VTT Electronics. 188 p. + app. 68 p. (VTT Publications 402). ISBN 951-38-5549-X; 951-38-5550-3. http://virtual.vtt.fi/inf/pdf/publications/1999/P402.pdf.

Niemelä, E. 2005. Strategies of product family architecture development. In: Obbink, H. & Pohl, K. (Eds.). Software Product Lines: Proceedings of the 9th International Conference, SPLC 2005. Rennes, France, 26–29 September 2005. Berlin, Heidelberg: Springer-Verlag. (Lecture Notes in Computer Science 3714) Pp. 186–197. ISBN 3-540-28936-4.

Niemelä, E., Kalaoja, J. & Lago, P. 2005. Toward an architectural knowledge base for wireless service engineering. IEEE Transactions on Software Engineering, Vol. 31, No. 5, pp. 361–379. ISSN 0098-5589.

Niemelä, E. & Matinlassi, M. 2005. Quality evaluation by QADA. A half-day tutorial in the 5th Working IEEE/IFIP Conference on Software Architecture, WICSA 2005. Pittsburgh, PA, USA, 6–9 November 2005.

Niemelä, E., Matinlassi, M. & Immonen, A. 2004. Practical evaluation of software product family architectures. In: Nord, R. (Ed.). Software Product Lines: Third international conference, SPLC 2004. Boston, MA, USA, 30 August – 2 September. New York: Springer Verlag. (Lecture Notes in Computer Science 3154). Pp. 130–145. ISBN 3-540-22918-3.

OMG. 2002. MOF 2.0 Query / Views / Transformations RFP. Request for Proposals, April 10th. 32 p. URL http://www.omg.org/docs/ad/02-04-10.pdf.

OMG. 2003a. MDA guide version 1.0.1. Miller, J. & Mukerji, J. (Eds.). Object Management Group, omg/2003-06-01. 12th June. 62 p.
URL http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf.

OMG. 2003b. UML profile for modeling quality of service and fault tolerance characteristics and mechanisms. Object Management Group, Revised submission, 94 p. URL http://www.omg.org/cgi-bin/apps/doc?ptc/05-05-02.pdf.

OMG. 2003c. UML profile for schedulability, performance, and time specification. Object Management Group, 235 p. URL http://www.omg.org/docs/formal/05-01-02.pdf.

OMG. 2004. MOF model to text transformation language. Request for proposals, August 2004. 31 p. URL http://www.omg.org/docs/ad/04-04-07.pdf.

OMG. 2005a. A proposal for an MDA foundation model. ormsc/05-04-01. An ORMSC White Paper V00-02, 7 p. URL http://www.omg.org/docs/ormsc/05-04-01.pdf.

OMG. 2005b. Unified modeling language 2.0: Superstructure. Object Management Group. 804 p. URL http://www.omg.org/docs/formal/05-07-04.pdf.

Perry, D. & Wolf, P. 1992. Foundations for the study of software architecture. ACM Sigsoft – Software Engineering Notes, Vol. 17, No. 4, pp. 40–52. ISSN 0163-5948.

Purhonen, A. 2002. Quality attribute taxonomies for DSP software architecture design. In: van der Linden, F. (Ed.). Proceedings of the 4th international workshop on software product-family engineering, PFE-4. Bilbao, Spain, 3–5 October 2001. New York: Springer. (Lecture Notes in Computer Science 2290.) Pp. 238–247. ISBN 3-540-43659-6.

Ramljak, D., Puksec, J., Huljenic, D., Koncar, M. & Simic, D. 2003. Building enterprise information system using model driven architecture on J2EE platform. In: Proceedings of the 7th international conference on telecommunications, ConTEL 2003. Zagreb, Croatia, 11–13 June 2003. Zagreb, Croatia: University of Zagreb. (Vol. 2.) Pp. 521–526.

Rodriques, G. N., Roberts, G. & Emmerich, W. 2004. Reliability support for the model driven architecture. In: de Lemos, R., Gacek, C. & Romanovsky, A.

Architecting Dependable Systems II. Berlin, Germany: Springer-Verlag. (Lecture Notes in Computer Science 3069.) Pp. 79–98. ISBN 3 540 23168 4.

Salicki, S. & Farcet, N. 2001. Expression and usage of the variability in the software product lines. In: van der Linden, F. (Ed.). Proceedings of the 4th international workshop on software product-family engineering, PFE-4. Bilbao, Spain, 3–5 October 2001. New York: Springer. (Lecture Notes in Computer Science 2290.) Pp. 287–297. ISBN 3-540-43659-6.

Selic, B. 2003. The pragmatics of model-driven development. IEEE Software, Vol. 20, No. 5, pp. 19–25. ISSN 0740-7459.

Selic, B. 2004. Model-Driven Development in the embedded environment with OMG Standards. Presentation in the second Summer School "MDA for Embedded Systems". Brest, Brittany, France, 6–10 September 2004.

Selic, B., Gullekson, G. & Ward, P. T. 1994. Real-time object oriented modeling. New York: Wiley. 525 p. ISBN 0-471-59917-4.

Sendall, S. & Kozaczynski, W. 2003. Model transformation: The heart and soul of model-driven software development. IEEE Software, Vol. 20, No. 5, pp. 42–45. ISSN 0740-7459.

Sha, L., Rajkumar, R. & Gagliardi, M. 1995. A software architecture for dependable and evolvable industrial computing systems. Pittsburgh, PA, USA: Software Engineering Institute. 24 p. (SEI Technical Reports CMU/SEI-95-TR-005.)

Shaw, M. 1995. Comparing architectural design styles. IEEE Software, Vol. 12, No. 6, pp. 27–41. ISSN 0740-7459.

Shaw, M. & Clements, P. 1996. Toward boxology: Preliminary classification of architectural styles. Proceedings of the 2nd International Software Architecture Workshop, ISAW-2. San Francisco, CA, USA, 14–15 October 1996. New York, NY, USA: ACM. Pp. 50–54.

Shaw, M. & Clements, P. 1997. Field guide to boxology: Preliminary classification of architectural styles for software systems. Proceedings of the

21st Annual International Computer Software & Applications Conference, COMPSAC'97. Washington, DC, USA, 13–15 August 1997. Los Alamitos, CA, USA: IEEE. Pp. 6–13. ISBN 0730-3157.

Sifakis, J., Tripakis, S. & Yovine, S. 2003. Building models of real-time systems from application software. Proceedings of the IEEE, Vol. 91, No. 1, pp. 100–111. ISSN 0018-9219.

Steenbergen, C., Rapella, D., Belaunde, M., Nektarious, G. & Tinella, S. 2004. Panel discussion: What is the added value of MDA for industry? The 1st European workshop on model-driven architecture with emphasis on industrial applications, MDA-IA 2004. University of Twente, Enschede, The Netherlands, 17–18 March 2004.
URL http://modeldrivenarchitecture.esi.es/ pdf/PanelDiscussion.zip

Taylor, R., Medvidovic, N., Anderson, K., Whitehead, E., Robbings, J., Nies, K., Oreizy, P. & Dubrow, D. 1996. A component- and message-based architectural style for GUI software. IEEE Transactions on Software Engineering, Vol. 22, No. 6, pp. 390–406. ISSN 0098-5589.

Törngren, M., Chen, D. & Crnkovic, I. 2005. Component-based vs. Model-based development: A comparison in the context of vehicular embedded systems. Proceedings of the 31st Euromicro conference on Software Engineering and Advanced Applications, Euromicro 2005. Porto, Portugal, 30 August – 3 September 2005. Los Alamitos, CA, USA: IEEE Computer Society. Pp. 432–441.

Vrijnsen, L., Delnooz, C., Somers, L. & Hammer, D. 2003. Experiences with scenario based architecting. Proceedings of the 16th international conference of software & systems engineering and their applications, ICSSEA 2003. Centre pour la Maîtrise des Systèmes et du Logiciel Conservatoire National des Arts et Mètiers, Paris, France, 2–4 December 2003.

Zhao, Q., Huang, G., Luo, X. & Wu, X. 2001. A software architectural style for deregulated power markets. Proceedings of 2001 Winter Meeting of the IEEE Power Engineering Society. Columbus, OH, USA, 28 January – 1 February 2001. Piscataway, NJ, USA: IEEE. (Vol. 3). Pp. 1497–1502. ISBN 0 7803 6672 7.

Appendices III and IV of this publication are not included in the PDF version.
Please order the printed version to get the complete publication
(http://www.vtt.fi/publications/index.jsp)

PAPER I

Comparison of software product line architecture design methods

COPA, FAST, FORM, KobrA and QADA

In: Proceedings of the 29th international conference on software engineering, ICSE2004. Edinburgh, Scotland, U.K., 23–28 May 2004. Los Alamitos, California: IEEE. Pp. 127–136.

Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA

Mari Matinlassi
VTT Technical Research Centre of Finland, P.O Box1100, 90571-Oulu FIN
Mari.Matinlassi@vtt.fi

Abstract

Product line architectures (PLAs) have been under continuous attention in the software research community during the past few years. Although several methods have been established to create PLAs there are not available studies comparing PLA methods. Five methods are known to answer the needs of software product lines: COPA, FAST, FORM, KobrA and QADA. In this paper, an evaluation framework is introduced for comparing PLA design methods. The framework considers the methods from the points of view of method context, user, structure and validation. Comparison revealed distinguishable ideologies between the methods. Therefore, methods do not overlap even though they all are PLA design methods. All the methods have been validated on various domains. The most common domains are telecommunication infrastructure and information domains. Some of the methods apply software standards; at least OMG's MDA for method structures, UML for language and IEEE Std-1471-2000 for viewpoint definitions.

1. Introduction

Software product lines (PL) are a well-known approach in the field of software engineering. Several methods have been published to address the problems of PL engineering. Methods are diverging in terminology and application domains. Therefore it is difficult to find out the differences and similarities of the methods.
Only few attempts have been made to evaluate or compare the product line architecture (PLA) design methods, e.g. in [1], [2] and [3]. Lopez-Herrejon and Batory propose a standard example case for evaluating product line methods. However, this example is very close to implementation and measures method features with performance benchmarking of the products the method outputs. This kind of evaluation of product line methods is very limited and a comparison covering also the other aspects of PL methods is required. The other example of surveys on product line architectures touches all the aspects related to the product line from assessment to domain engineering and testing. However, this report either does not provide any comparisons that would concern product line design methods. The third attempt represents a covering survey on software architecture *analysis* methods however, software architecture *design* methods are not considered.

On the basis of our studies, there are five methods answering the needs of product lines from the software architectural point of view. In alphabetical order they are COPA[4], FAST[5], FORM[6], KobrA[7] and QADA[8].

The first of the methods mentioned, a Component-Oriented Platform Architecting Method for product family engineering, i.e. COPA, is a component-oriented but architecture-centric method that enables the development of software intensive product families. FAST – Family-Oriented Abstraction, Specification and Translation - is a software development process focused on building families. Feature-Oriented Reuse Method for product line software engineering, FORM is an extension to the FODA [9] method. The core of FORM lies in the analysis of domain features and the use of these features to develop reusable and adaptable domain artifacts. That is, FORM is a feature-oriented approach to product line architecture engineering. Kobra again is an acronym for Komponentenbasierte Anwendungsentwicklung, denoting a practical method for component-based product line engineering with UML. Quality-driven Architecture Design and Analysis, shortly QADA states a product line architecture design method providing traceable product quality and design time quality assessment.
The purpose of this investigation was to study and compare the existing methods for the design of software product line architectures. The intention of this paper is not to provide an exhaustive survey on the area but provide a state-of-the-art of current PLA practices and help others to understand and contrast alternative approaches to product line design. This paper does neither guide in selecting the right approach for PLA design but opens up a basis for creation of such a decision tool. First, this paper provides background knowledge on architectural design methods and introduces a comparison framework for evaluating PLA design methods. Then, the five PLA design methods are briefly presented and compared against the framework. The most remarkable observations of the comparison close the paper.

2. Architecture Design

Architectural views have been the basis for a number of techniques developed and used during the last few years for describing architectures. It seems that the first of them was "4+1 views to software architecture" [10]. The four main views used are logical, process, physical and development view. The logical view describes an object model. The process view describes the design's concurrency and synchronization aspect. The physical view describes the mapping of the software onto the hardware reflecting the distributed aspect of the system. The development view describes the software's static organization in its development environment. The '+1' denotes the use-case view consisting of scenarios that are used to illustrate the four views.

Jaaksi et al. [11] suggests a slightly modified version of the 4+1 view technique and ends up with 3+1 views necessary to describe the software architecture. The views are the logical, runtime and development view, plus the scenario view. The 3+1 method applies the Unified Modeling Language (UML) as an architectural description language.

Hofmeister et al. [12] define four views (conceptual, module, execution and code view) that are based on observations done in practice on various domains, e.g. image and signal processing systems, a real-time operating system, communication systems, etc.

Despite the fact that the techniques introduced above are capable and exhaustive in their own way; none of them concerns the product line approach to the architectural design.

Architecture Based Design (ABD) method [13] is a quality driven method for designing the software architecture for a long-lived system at the conceptual level of abstraction. In ABD, the conceptual architecture is a representation of the high-level design choices described with three architectural views. Even though the ABD method has been developed further into a new method called the Attribute Driven Design method, ADD [14], it still does not provide more than a coarse grained high-level, i.e. conceptual architecture as an output. Also the support for product line architecture design in the ABD and in the ADD is mentioned but immature.

Only methods (1) specialized for architecture engineering of software product lines and (2) with sufficient materials were selected for comparison. The product line practices concerned e.g. in [15], namely at least Synthesis, Sherlock and Odyssey-DE, are out of the range of this investigation. In addition, QASAR by Bosch [16] describes the process and lists method artifacts leaving the other aspects of the method hidden. SPLIT by Coriat et al. [17] also has insufficient materials and is out of the scope of the evaluation.

3. An Evaluation Framework

An evaluation framework that is introduced in Table 1 is used as an analysis tool. The framework is based on three sources. The first is the NIMSAD (Normative Information Model-based Systems Analysis and Design) evaluation framework [18]. NIMSAD framework uses the entire problem solving process as the basis of evaluation and it can be used to evaluate methods on any category. According to NIMSAD, there are four essential elements for method evaluation.

Firstly, the method is evaluated from the element of the problem situation, i.e. the method context. The second element is the intended problem solver, i.e. the user of the method. The third element is the problem solving process, i.e. the method itself. The last element brings the three elements together through self-evaluation. Because rare methods consider evaluation of the method context, or user or contents, herein, the method evaluation element is turned to method validation element and it considers the validation of the method in question and validation of method outputs.

In addition to the NIMSAD framework, the definition of a method and its ingredients [19] has influenced the third element of the framework, i.e. the method contents. Kronlöf defines method ingredients as follows: 1) an underlying model, 2) a language, 3) defined steps and ordering of these steps and 4) guidance for applying the method. Because tools help in execution of the methods, they are also considered in the element of method contents. The third source for the evaluation framework is an application of the NIMSAD framework for component-based software development methods [20].

The goal of this evaluation was not to rate the methods but to provide an overview of current PLA engineering methods and find out if - and how - the methods differ in any aspects of the PLA design. Therefore a neutral, common and quite extensive NIMSAD framework for method evaluation was utilized to derive the fundamental element categories for the framework. NIMSAD framework has earlier been applied in evaluation of software engineering methods. This application of the framework on software engineering methods provided the basis for detailed element definition for categories. With various questions this study tries to address e.g. maturity, practicality and scope of the methods to find differences. On the other hand the goal was to study if the methods really have what it takes to call them a method. These elements were considered in the category of 'contents' by questioning if the methods satisfy the definition of a method. Framework elements were refined to cover features special for product line methods (e.g. variability support). Herein, the evaluation of the "artifact" element is excluded because of space limitations.

Table 1. The categories and elements of the framework and the questions used in the evaluation.

Category	Elements	Questions
Context	Specific goal	What is the *specific* goal of the method?
	Product line aspect(s)	What aspects of the product line does the method cover?
	Application domain(s)	What is/are the application domain(s) the method is focused on?
	Method inputs	What is the starting point for the method?
	Method outputs	What are the results of the method?
User	Target group	Who are the stakeholders addressed by the method?
	Motivation	What are the user's benefits when using the method?
	Needed skills	What skills does the user need to accomplish the tasks required by the method?
	Guidance	How does the method guide the user while applying the method?
Contents	Method structure	What are the design steps that are used to accomplish the method's specific goal?
	Artifacts	What are the artifacts created and managed by the method?
	Architectural viewpoints	What are the architectural viewpoints the method applies?
	Language	Does the method define a language or notation to represent the models, diagrams and other artifacts it produces?
	Variability	How does the method support variability expression?
	Tool support	What are the tools supporting the method?
Validation	Method maturity	Has the method been validated in practical industrial case studies?
	Architecture quality	How does the method validate the quality of the output it produces?

4. Overview of PLA design methods

4.1. COPA

A Component-Oriented Platform Architecting Method for Families of Software Intensive Electronic Products (COPA) is being developed at the Philips Research Labs. The COPA method is one of the results of the Gaudi project [21]. The ambition of the Gaudí project is "to make the art and emerging methodology of System architecture more accessible and to transfer this know how and skills to a new generation of system architects".

The specific goal of the COPA method is to achieve the best possible fit between business, architecture, process and organization. This goal results in the middle name of the COPA method: the BAPO product family approach [22]. The specific goal of architecture design is to find a balance between component-based and architecture-centric approaches [23], wherein the component-based approach is a bottom-up approach relying on composition. The architecture-centric approach is a top-down approach relying on variation.

COPA covers the following aspects of product lines: business, architecture, process and organizational aspects. Herein, our evaluation concentrates on the architecture and process aspects. According to [24], the application domains of the COPA method are telecommunication infrastructure systems and the medical domain. In addition, the case studies on the consumer electronics domain are discussed in [4] and [25]. Within these domains, COPA assists in building product populations [23]. Product populations denote the large-scale diversity in a product family developed with a component-driven, bottom-up, partly opportunistic software development using, as much as possible, available software to create products within an organization.

Originally, the COPA method starts by analyzing the customer needs. To be more specific, the inputs of the method's architecting phase are facts, stakeholder expectations, (existing) architecture(s) and the architects(s) intuition. The completely applied COPA method produces the final products. To be more specific, the phase of "architecting" aims to produce a lightweight architecture (see [26] for definition) as an output. A lightweight architecture denotes guidelines for architecture more than traditional software decomposition.

COPA is an extensive method targeted to all interest groups of a software company. Especially, the architecture stakeholders of the COPA method are the customers, suppliers, business managers and engineers [27]. The "multi-view" architecting is addressed for these four main stakeholders [26]. Motivation to use COPA is a promise to manage size and complexity, obtain high quality, manage diversity and obtain lead time reduction.

4.2. FAST

David Weiss introduced a practical, family-oriented software production process in the early 1990's. The process is known as the Family-Oriented Abstraction, Specification, and Translation process. At the time of writing the book on FAST (1999), the process was in use at Lucent Technologies and the evolution was continuing. The FAST [5] process is an alternative to the traditional software development process. It is applicable wherever an organization creates multiple versions of a product that share significant common attributes, such as common behavior, common interfaces, or common code.

The specific goal of FAST is to make the software engineering process more efficient by reducing multiple tasks, by decreasing production costs, and by shortening the marketing time.

Considering the product line aspects, the FAST method defines a full product line engineering process with activities and artifacts. FAST divides the process of a product line into three sub processes, i.e. domain qualification, domain engineering and application engineering [5].

FAST has been applied successfully at Lucent Technologies at least on the domains of telecommunication infrastructure and real-time systems.

Domain qualification starts from receiving the general needs of business line by distinguishing between two cases: one in which you pay little or no attention to domain engineering, and a second one in which you engineer the domain with the intent of making production of family members more efficient. Application engineering starts when application engineers receive the requirements from customers.

Domain qualification outputs an economic model to estimate the number and value of family members and the cost to produce them. Domain engineering generates a language for specifying family members, an environment for generating family members from their specifications, and a process for producing family members using the environment. Application engineering generates family members in response to customer requirements as an output.

The FAST method was born in the industry and has a high practical background. Therefore, FAST seems to be aimed at software engineers and designers currently working in the industry. The use of the FAST method is motivated with a desire to alleviate the problems that make the software developers' task a lengthy and costly one.

4.3. FORM

Kyo C. Kang and his co-fellows in Pohang University of Science and Technology, Korea, propose a Feature-Oriented Reuse Method (FORM) [6] as an extension to the Feature-Oriented Domain Analysis (FODA) method [9]. FORM extends FODA to the software design and implementation phases and prescribes how the feature model is used to develop domain architectures and components for reuse.

FORM has a specific goal on how to apply domain analysis results (commonality and variability) to the engineering of reusable and adaptable domain components with specific guidelines.

The application domain(s) for the FORM method are the telecommunication domain and the information domain. However, the feature exists in any application domain. If the feature model can be obtained from the application domain, FORM can be fit to the needs of other specific domains.

FORM starts with feature modeling to discover, understand, and capture commonalities and variabilities of a product line. Domain engineering starts from the beginning of the software development: context analysis. The primary input is the information on systems that share a common set of capabilities and data.

Domain engineering creates the feature model, reference architecture, and reusable components as an output. Application engineering creates the application software after features have been selected from the feature model, application architecture has been selected from reference architecture and reusable components have been selected from reusable components.

FORM is targeted to the wide spectrum of domain and application engineering, including the development of reusable architectures and code components. It is used at software engineering in many industrial aspects.

The model that captures commonalities and differences is called a "feature model". The use of features is motivated by the fact that customers and engineers often speak of product characteristics in terms of "features the product has and/or delivers". Features are abstractions that both customers and developers understand and should be the first class objects in software development.

4.4. KobrA

Fraunhofer IESE has been developing the KobrA method [28], [29], [30] that is a methodology for modeling architectures. The method stands for Komponentenbasierte Anwendungsentwicklung that is German for "component-based application development" [7].

KobrA denotes itself as a component-based incremental product line development approach or a methodology for modeling architectures. It is also designed to be suitable for both single system and family based approaches in software development. In addition, the approach can be viewed as a method that supports a Model Driven Architecture (MDA) [31] approach to software development, in which the essence of a system's architecture is described independently of platform idiosyncrasies. Another important goal is to be as concrete and prescriptive as possible and make a clear distinction between the products (i.e. artifacts) and processes.

KobrA defines a full product line engineering process with activities and artifacts. The most important parts of PL engineering are framework engineering and application engineering with their sub steps, but KobrA also defines implementation, releasing, inspection and testing **aspects** of product line engineering process.

KobrA is developed for the information systems domain (i.e. library system in [32]). However, different application domains demand different methodical support. Therefore, KobrA can be customized to better fit the needs of a specific project. The method provides support for being changed in terms of its processes and products. In addition to the application domain, the factors influencing the KobrA method are organizational context, project structure and the goals of the project.

Framework engineering starts from the very beginning of the software development: context realization. Framework engineering does not need any other input than the idea of a new framework with two or more applications.

The other main activity of the method - application engineering - starts when a customer contacts the software development organization. When such an expression of interest is received, an application engineering project is set up and the context realization instantiation is initiated. This activity equals to the elicitation of user requirements within the scope of the framework.

'Komponent realizations' mean low level designs of software components. However, the process is defined as far as to the implementation and testing phases of the software product.

KobrA is definitely aimed at software engineers and designers currently working in the industry. It is a simple method for developing software and the adoption of the method does not probably express overwhelming challenges for software practitioners today. It also provides an opportunity to get involved in the development of a family of applications and smoothly encourages seeking the benefits of reusing existing assets.

KobrA states it is a simple, systematic, scalable and practical method [7]. Simple here means that a method is as economic as possible with its concepts and the features in a method should be as orthogonal as possible. In addition, a method should separate concerns to the greatest extent possible. Systematic expects that the concepts and guidelines defined in the method should be precise and unambiguous. Also, a method should tell developers what they should do, rather that what they may do. Another feature of the method, that products of a method are strictly separated from the process, also serves in reaching a systematic method. A scalable method provides two aspects of scalability, these being granularity scalability and complexity scalability. The first one means that a method should be able to accommodate large-scale and small-scale problems in the same manner using the same basic set of concepts, whereas fulfillment of the last one refers to incremental application of the method concepts. Practicality requires that a method is compatible with as many commonly used implementation and middleware technologies as possible, particular those that are either de facto or de jure standards.

4.5. QADA

The QADA method is being developed at VTT, the Technical Research Centre of Finland. QADA is an abbreviation for Quality-driven Architecture Design and quality Analysis, a method for both to design and to evaluate software architecture of service-oriented systems.

QADA claims to be a quality-driven architecture design method. It means that quality requirements are the driving force when selecting software structures and, each viewpoint concerns certain quality attributes [33]. Architecture design is combined with quality analysis, which discovers if the designed architecture meets the quality requirements set in the very beginning.

QADA method describes the architectural design part of the software development process, including steps and artifacts produced in each step. It also covers the description language used in the artifacts. It does not cover organizational or business aspects.

Quality-driven design is aimed for middleware and service architecture domains. The case studies cover the design of distributed service platform [8], two kinds of platform services for wireless multimedia applications [34] and the design of wireless multimedia game [35]. In addition, a recent case study on traffic information management system is mentioned in [36]. Quality analysis has been applied to the middleware platform [8], spectrometer controller [37] and terminal software [36].

The method starts with the requirements engineering phase that – even though called requirements engineering - means a link between requirements engineering and software architecture design. The aim is in collecting the "driving ideas of the system and the technical properties on which the system is to be designed" [8]. In addition to functional properties, the quality requirements and constraints of the system are captured as input.

The output of the QADA method is twofold: design and analysis. Design covers software architecture at two abstraction levels: conceptual and concrete. Conceptual architecture covers the conceptual components, relationships and responsibilities, which are intended to be used by certain high level stakeholders related to product line, e.g. product line architects or management. Concrete architecture is closer to the so-called 'traditional' architecture description aimed for software engineers and designers. The QADA method does not produce implementation artifacts.

Analysis provides precious information concerning the quality of the design. Analysis results in feedback of whether the design addresses the quality requirements defined for the system. Analysis may also produce quality feedback about an existing system.

The method users are product line architects and software architects or an architecting team. However, the group of stakeholders that use the method output is much wider. At the conceptual level, the stakeholders include system architects, service developers, product architects and developers, maintainers, component designers, service users, project manager and component acquisition, whereas at concrete level, the architectural descriptions are aimed at component designers, service developers, product developers, testing engineers, integrators, maintainers and assets managers. These groups continue by implementing, testing or maintaining the architecture that is designed.

QADA claims - as do almost all the methods – to be a systematic method and simple to learn. In addition, it is applicable to existing modeling tools [8]. The architecture modeling method also improves communication among various stakeholders [38] and conforms to the IEEE standard for architectural description [39].

5. Comparison Results

5.1. Context

Each of the methods under evaluation is distinguishable concerning the specific goal the method has. All the methods have the same *overall goal*, i.e. produce product line architectures. However, to find a difference, a *specific goal* denotes what point(s) does the method press or highlight in PLA development.

Although e.g. both COPA and KobrA are component-based, the COPA method stands out by combining component-based (i.e. bottom-up) and architecture-centric (top-down) approaches with a novel way. Another top-down approach in addition to COPA is the QADA method. However again, QADA has a diverging goal in combining quality-driven approach with the architecture-centric one. The FAST method expresses itself as a process-driven method, and finally, the FORM method represents well-known feature-orientation to product line engineering.

As a feature-oriented approach, FORM states that it also covers the requirements engineering. The commonality analysis in the FAST method covers the requirements phase extensively. The other methods seem to step aside in this area, except that the QADA method represents an interface between requirements engineering and architecture design however this interface cannot be considered as a systematic approach to gather and analyze product requirements. In addition to requirements engineering, the FORM method covers architecture, implementation and process, as does also the KobrA method. What comes to the other methods, the COPA method is the most complete, covering all the aspects of a product line, whereas FAST captures only the process aspect and QADA extends the method's scope from process aspects to architectural aspects.

The information systems domain is the most popular application domain; three methods altogether, namely KobrA (library system [32]), FORM (electronic bulletin board [40]) and QADA (traffic information management system [36]). In addition to the information system, QADA has been applied in middleware [8], [34], the wireless multimedia domain [35] and in the space application domain [37].

In addition to the electronic bulletin board system, the FORM method has been applied on the elevator control system [41] and the telecommunication infrastructure system [42]. The telecommunication infrastructure domain has been the application domain of also COPA [24] and FAST [5]. The FAST method has been applied on the domain of real-time systems as well [5]. Quite apart from that, the COPA method alone among the methods extends to the medical domain. The COPA case studies on the consumer electronics domain are discussed in [4], [25].

All the methods start from the very beginning, taking context or user requirements as input. While considering the method outputs, all the methods seem to produce quite in-depth outputs by generating results that are close to the implementation. COPA also takes a wider insight into the issue by considering the business and organizational aspects. KobrA defines the process as far as to the implementation and testing phases of the software product. Furthermore, the QADA method is distinguished with output information concerning the quality of the design.

5.2. User

The users of the method are either people who actually use the method, i.e. follow the steps and create the defined artifacts, or people who benefit and use the outputs of the method. It seems the methods agree on the rough division of stakeholder groups related to product line engineering: engineers, architects, business managers and customers. To make a difference, KobrA perhaps is the most practical method aimed at software engineers and designers currently working in the industry. It is a simple method for developing software, and the adoption of the method does not probably express overwhelming challenges for software practitioners today. The conformance to a language standard (UML) and usage of commercial tools emphasizes the practicality and applicability of the KobrA method. Quite the contrary one may say that FORM is aimed at the academic audience.

What comes to the motivation, adopting any of these product line architecture design methods provides several benefits e.g. reuse, complexity management, higher quality and shorter time-to-market. However, these benefits do not motivate the real method users (software architects) as well as the following implicit reasons. Both KobrA and QADA are developed with a goal to produce a simple and systematic method. They also conform to commonly known standards: UML (KobrA), MDA (KobrA [7] and QADA [38]) and IEEE-Std-1471-2000 (QADA [33]). With an industry proven background COPA is a practical method, and with extensive architectural descriptions, it improves communication among various stakeholders of PL engineering. As well, feature-orientation of FORM gives a common language and therefore improves communication between customers and engineers.

Considering the question of what are the skills the method users need when applying the method, the following issues were concluded. One of the essential method properties is the method language. Two of the methods have a special notation language or ADL to learn (see [6] for FORM notation and COPA Koala [43]) and the most of the methods apply UML as description language. However, current commercial UML tools do not provide a sufficient customization aspect to the needs of architectural descriptions and therefore, every one of the methods need special or extended tool support. This will scale up the effort needed to learn the method. Furthermore, each method has its own method ideology needed to learn. However a 'skill' needed for this purpose is just an open mind.

All the methods provide descriptive case studies. In addition, FORM provides a special guideline [44] for using a feature-oriented approach. COPA and QADA suffer a lack of method documentation, whereas FAST and KobrA are captured in extensive manuals.

5.3. Contents

FORM, FAST and KobrA define a quite similar structure for the method. The basic idea is to first define the context of the system. After that the main two phases are (1) domain engineering and (2) application engineering. Domain engineering is also called product family engineering or framework engineering and it analyses the commonalities and variabilities among requirements and defines the domain architecture or a component framework. Application engineering instantiates the architectural model from domain architecture and produces application realization. In addition to these two main phases, the COPA method introduces the third phase called platform engineering. Platform engineering focuses on the development, maintenance and administration of reusable assets within the platform. Therefore, platform engineering is nothing more than a sub phase derived from domain engineering. Despite, the steps defined in the QADA method are diverging. First, an interface is defined for requirements engineering, which is somewhat compliant to the context analysis. However, design is divided into two phases of conceptual and concrete architecture design. After both design phases, QADA introduces the phase of quality evaluation that assesses the quality of architectural design against defined quality attributes.

FORM and FAST explicitly define support for variability in requirements elicitation, whereas the other methods do not. In addition, through tool support [45] FORM provides automatic transformation from the requirements to an instance of the domain architecture. The other methods concentrate on capturing variability with graphical language in architectural design. QADA and KobrA content themselves with adapted UML and manual transformation to code, whereas COPA has developed its own language and tools to represent variability and transform component descriptions automatically into code skeletons.

FAST does not define explicit tool support. Instead, Process and Artifact State Transition Abstraction (PASTA) process modeling tool of FAST serves to explain FAST in more detail, to help the user to improve FAST and to help the user to develop automated support for FAST. Quite contrary, the FORM method has a single tool, ASADAL [45] supporting all the features mentioned in [40]. Concerning the rest of the methods, they all mention a set of tools (Table 2).

Table 2. Comparing sets of tools in COPA, KobrA and QADA.

Tool	Method		
	COPA	KobrA	QADA
Koala compiler (special code generator)	X		
KoalaMaker (produces a makefile)	X		
Commercial code editor	X		
Commercial UML tool		X	X
Plug-ins for code editor	X		
Visio (with special stencil)	X		X
Word processing tool		X	X
Configuration management		X	

Only two of the methods (COPA and QADA) apply views/viewpoints. Also FORM mentions three architectural models (also called viewpoints): subsystem, process and module [6]. However, an architectural view/viewpoint [39] is a far broader concept than just a model and closely related to various stakeholders of PLA.

The COPA method refines architecture into five views: customer, application, functional, conceptual and realization views [24], [46]. When looking at the view descriptions (Table 3) it is seen that COPA views are more oriented on describing the whole product than just software architecture. The first two views are so called “commercial” views and the last two are technical views. The Functional view in the middle is both commercial *and* technical [27]. However, “no attempt has been made yet to map out the collected viewpoints on the IEEE [39] ontology [27]“. Instead, QADA viewpoints (Table 4) conform to the IEEE standard viewpoint description and provide various viewpoints on the software architecture of the system. The four viewpoints are provided at two levels of abstraction. The difference between the levels is partly also in the aggregation dimension (see [47] for definitions of architectural dimensions). Abstraction level means both abstractions with respect to the main architectural concepts of the system and abstraction from the physical world. Therefore a component at the conceptual level is not a software component but more like a logical concept.

Table 3. Introducing COPA views.

View	Description
Customer	Describes the customer’s world. Business modeling from the customer’s viewpoint.
Application	Describes the applications that are important to the customer. Application modeling. Customer ‘how?’
Functional	Captures the system requirements of a customer application. Product ‘what?’
Conceptual	Includes the architectural concepts of the system. Product ‘how?’ Component identification and aspect design
Realization	Describes the realization technologies for building the system. Product ‘how?’ “Implementation is part of the realization view [48]”

Table 4. Introducing QADA viewpoints.

Viewpoint	Description
Structural	Structures involved in particular functional or/and quality responsibilities. Quality analysis [36].
Behavior	Dynamic actions of, and within a system, their ordering and synchronization. Analysis of execution qualities [36].
Deployment	Structures are deployed into processes and/or physical computing units. Analysis of execution qualities.
Development	Organizing the design work, describes the technological choices made upon standards, software realization asset management

5.4. Validation

All of the methods have been validated in practical industrial case studies. The COPA method was born in the industry and therefore, perhaps, has the strongest industrial experience with software applications in large product families.

Most of the methods i.e. FORM, FAST, COPA and KobrA ensure quality attributes with non-architectural evaluation methods, such as model checking, inspections and testing. Although KobrA also proposes scenario-based architecture evaluation (SAAM [49]) for ensuring maintainability, none of these methods define an explicate way to validate the output from the domain of application engineering. Despite this, the QADA method has an exceptional way of evaluating software architecture designs before implementation. The quality of the design is validated with a scenario based evaluation method in two phases: conceptual and concrete [8].

6. Conclusions

This study has compared five methods for product line architectural design: COPA, FAST, FORM, KobrA and QADA according to specially developed question framework. The comparison largely rested on the available literature. Based on the combined experience of the five product line engineering methods, the most important conclusions were as follows.

The methods do not seem to compete with each other, because each of them has a special goal or ideology. All the methods highlight and follow this ideology throughout the method descriptions.

- COPA. Concentrated on balancing between top-down and bottom-up approaches and covering all the aspects of product line engineering i.e. architecture, process, business and organization.
- FAST. Family oriented process description with activities, artifacts and roles. Therefore, it is very adapting but not applicable as it is.
- FORM. Feature-oriented method for capturing commonality inside a domain. Extended also to cover architectural design and development of code assets.
- KobrA. Practical, simple method for traditional component-based software engineering with UML. Adapts to both single systems and family development.
- QADA. Concentrated on architectural design according to quality requirements. Provides support for parallel quality assessment of product line software architectures.

The most popular domains for applying the methods have been information and telecommunication (infrastructure) domains. These domains have six case studies published all together. However, also the real-time domain, wireless services, middleware, medical systems and consumer electronics domains have been on trial.

All the methods agree that none of the available commercial tools alone and/or without extensions support product line architectural design. Therefore, special tools or tool extensions have been developed to form a set of tools. This way, product line methods may have a full, practical tool support.

There are not available de jure standards for product line architecture development. KobrA and QADA apply other software standards - namely OMG MDA and UML and IEEE Std-1471-2000 – which provide support for formalizing PLA design.

The aim of this study was to provide a comparative analysis and overview on the PLA engineering methods. In addition, this study may provide a basis for developing a decision tool for selecting an appropriate PLA engineering practice. Meanwhile, getting familiar with all the approaches before embarking on suitable PLA development method is recommended.

Acknowledgements
I want to thank Philips, Avaya Labs, Pohang University, IESE and VTT on their constructive criticism. I also appreciate the help of Mr. Hailang Zuo in data collecting and want to thank Professor Eila Niemelä for providing valuable comments during the work.

7. References

[1] R. Lopez-Herrejon and D. Batory, "A Standard Problem for Evaluating Product-Line Methodologies," in the *Proc. of Generative and Component-based Software Engineering: Third International Conference, GCSE 2001*, vol. 2186, *Lecture Notes in Computer Science*. Springer Verlag, Berlin Heidelberg, 2001.

[2] M. Harsu, "A Survey of Product-Line Architectures," Tampere University of Technology, Report 23, March 2001.

[3] L. Dobrica and E. Niemelä, "A Survey on Software Architecture Analysis Methods," *IEEE Transactions on Software Engineering*, vol. 28, 2002, pp. 638-653.

[4] P. America, H. Obbink, J. Muller, and R. van Ommering, "COPA: A Component-Oriented Platform Architecting Method for Families of Software Intensive Electronic Products,". Denver, Colorado: The First Conference on Software Product Line Engineering, 2000.

[5] D. Weiss, C. Lai, and R. Tau, *Software product-line engineering: a family-based software development process*. Addison-Wesley, Reading, MA, 1999.

[6] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures," *Annals of Software Engineering*, vol. 5, 1998, pp. 143 - 168.

[7] C. Atkinson et al., *Component-based product line engineering with UML*. Addison-Wesley, London, New York, 2002.

[8] M. Matinlassi, E. Niemelä, and L. Dobrica, "Quality-driven architecture design and quality analysis method, A revolutionary initiation approach to a product line architecture," VTT Technical Research Centre of Finland, Espoo, 2002.

[9] K. C. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis. Feasibility study,," Software Engineering Institute, Pittsburgh CMU/SEI-90-TR-21, 1990.

[10] P. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, vol. 12, 1995, pp. 42-50.

[11] A. Jaaksi, J.-M. Aalto, A. Aalto, and K. Vättö, *Tried & True Object Development: Industry-Proven Approaches with UML*. Cambridge University Press, Cambridge Univ., 1999.

[12] C. Hofmeister, R. Nord, and D. Soni, *Applied Software Architecture*. Addison-Wesley, Reading, MA, 2000.

[13] F. Bachmann, L. Bass, G. Chastek, P. Donohoe, and F. Peruzzi, "The Architecture Based Design Method," CMU/SEI, Technical report 2000-TR-001, 2000.

[14] L. Bass, M. Klein, and F. Bachmann, "Quality Attribute Primitives and the Attribute Driven Design Method," in *4th International Workshop on Software Product-Family Engineering*, F. van der Linden, Ed. Springer, Berlin Heidelberg, 2002, pp. 163 - 176.

[15] C. Kuloor and A. Eberlein, "Requirements Engineering for Software Product Lines," in the *Proc. of the 15th International Conference of Software and Systems Engineering and their Applications (ICSSEA'2002)*, vol. 1. Conservatoire National de Arts et Métiers, Paris, 2002.

[16] J. Bosch, *Design and use of software architectures: adopting and evolving a product-line approach*. Addison-Wesley, Harlow, 2000.

[17] M. Coriat, J. Jourdan, and F. Boisbourdin, "The SPLIT Method, Building Product Lines for Software-Intensive Systems," in the *Proc. of the First Software Product Lines Conference*, P. Donohoe, Ed. Kluwer Academic Publishers, Boston, 2000, pp. 147 - 166.

[18] N. Jayaratna, *Understanding and evaluating methodologies: NIMSAD: a systematic framework*. McGraw-Hill, London, 1994.

[19] K. Kronlöf, *Method Integration: Concepts and Case Studies*. John Wiley & Sons, Chichester, 1993.

[20] M. Forsell, V. Halttunen, and J. Ahonen, "Evaluation of Component-Based Software Development Methodologies," in *Proceedings of FUSST'99*, J. Penjan, Ed. Institute of Cybernetics at TTU, Tallinn, 1999.

[21] Philips, http://www.extra.research.philips.com/natlab/sysarch/GaudiProject.html

[22] R. van Ommering, "Building Product Populations with

Software Components," in *Proceedings of ICSE'02*. ACM, 2002, pp. 255 - 265.

[23] R. van Ommering and J. Bosch, "Widening the Scope of Software Product Lines - From Variation to Composition," in *The Proc. of the Second Product Line Conference, SPLC2*, vol. 2379, *Lecture Notes in Computer Science*, G. Chastek, Ed. Springer-Verlag, Berlin, Heidelberg, 2002, pp. 328 - 347.

[24] J. Wijnstra, "Critical Factors for a Successful Platform-Based Product Family Approach," in the *Proc. of the Second Product Line Conference SPLC2*, vol. 2379, *Lecture Notes in Computer Science*, G. Chastek, Ed. Springer-Verlag, Berlin Heidelberg, 2002, pp. 68 - 89.

[25] R. van Ommering, "Building Product Populations with Software Components," in the *Proc. of the ICSE'02*. ACM, 2002, pp. 255 - 265.

[26] G. Muller, "Light Weight Architecture: the way of the future?," Embedded Systems Institute, Article written as part of the Gaudí project 18th March 2003.

[27] G. Muller, "A Collection of Viewpoints," Philips Research, 2001.

[28] C. Atkinson, J. Bayer, and D. Muthig, "Component-Based Product Line Development. The KobrA Approach," in the *Proc. of the First Software Product Lines Conference (SPLC1).* P. Donohoe, Ed. Kluwer Academic Publishers, Boston, 2000, pp. 289 - 309.

[29] C. Atkinson, J. Bayer, O. Laitenberger, and J. Zettel, "Component-based Software Engineering: The KobrA Approach," 22nd International Conference on Software Engineering (ICSE2000), International Workshop on Component-Based Software Engineering, Limerick, Ireland, June 5-6, 2000 2001.

[30] C. Atkinson and D. Muthig, "Component-based product-line engineering with the UML (tutorial)," in the Proc. of the 7th International Conference on Software Reuse, Berlin2002.

[31] D. Frankel, *Model Driven Architecture, Applying MDA to Enterprise Computing*. Wiley Publishing Inc., Indianapolis, Indiana, 2003.

[32] J. Bayer, D. Muthig, and B. Göpfert, "The Library System Product Line - A KobrA Case Study," Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Technical Report IESE-Report No. 024.01/E, 2001.

[33] A. Purhonen, E. Niemelä, and M. Matinlassi, "Viewpoints of DSP Software and Service Architectures," *Journal of Systems and Software*, vol. 69, 2004, pp. 57 - 73.

[34] A. Tikkala and M. Matinlassi, "Platform services for wireless multimedia applications: case studies," in the *1st International Conference on Mobile and Ubiquitous Multimedia*, Oulu, Finland, 2002, pp. 76 - 81.

[35] P. Lago and M. Matinlassi, "The WISE Approach to Architect Wireless Services," in *Proceedings of the 4th International Conference in Product Focused Software Process Improvement, PROFES2002*, *Lecture Notes in Computer Science*, M. Oivo and s. Komi-Sirviö, Eds. Springer, Berlin, Heidelberg, 2002, pp. 367 - 382.

[36] M. Matinlassi and E. Niemelä, "The Impact of Maintainability on Component-based Software Systems," in the *Proc. of the 29th Euromicro Conference*. IEEE Computer Society, Antalya, Turkey, 2003, pp. 25 - 32.

[37] Dobrica Liliana and N. Eila, "Attribute-based product-line architecture development for embedded systems," in the *Proc. of the 3rd Australasian Workshop on Software and Systems Architectures*. IEEE, Sydney, 2000, pp. 76 - 88.

[38] M. Matinlassi and J. Kalaoja, "Requirements for Service Architecture Modeling," in *Workshop of Software Modeling Engineering of UML2002*. Dresden, Germany, 2002.

[39] IEEE, "IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems," *Std-1471-2000*. New York: Institute of Electrical and Electronics Engineers Inc., 2000.

[40] K. C. Kang, "A Feature-Oriented Method for Product Line Software Engineering," Denver, Colorado: The First Software Product Lines Conference, 2000.

[41] K. Lee, K. C. Kang, E. Koh, W. Chae, B. Kim, and B. W. Choi, "Domain-Oriented Engineering of Elevator Control Software: A Product Line Practice," in *Software Product Lines, Experience and Research Directions*, P. Donohoe, Ed. Kluwer Academic Publishers, Boston, 2000, pp. 3 - 22.

[42] K. C. Kang, S. Kim, J. Lee, and K. Lee, "Feature-Oriented Engineering of PBX Software for Adaptability and Reusability," *Software Practice and Experience*, vol. 29, 1999, pp. 875 - 896.

[43] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala component model for consumer electronics software," *IEEE Computer*, vol. 33, 2000, pp. 78 - 85.

[44] K. Lee, K. C. Kang, and J. Lee, "Concepts and Guidelines of Feature Modeling for Product Line Software Engineering," in the *Proc. of the 7th International Conference on Software Reuse*, *LNCS 2319*, C. Gacek, Ed. Springer-Verlag, Berlin Heidelberg, 2002, pp. 62 - 77.

[45] ASADAL, http://selab.postech.ac.kr/realtime/public_html/index.html

[46] P. America, H. Obbink, and E. Rommes, "Multi-View Variation Modeling for Scenario Analysis," in *PFE-5: Fifth International Workshop on Product Family Engineering*, F. van der Linden, Ed. Springer, 2003.

[47] L. Bratthall and P. Runeson, "A Taxonomy of Orthogonal Properties of Software Architecture," in the *Proc. of the Second Nordic Software Architecture Workshop*, 1999.

[48] G. Muller, "Software Reuse; Caught between strategic importance and practical feasibility," Embedded Systems Institute, Article as part of the Gaudí project, 19th March 2003.

[49] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley, Reading, Massachusetts, 1998.

PAPER II

Quality-driven architecture design method

In: Proceedings of the 15th international conference of software & systems engineering and their applications, ICSSEA 2002. Centre pour la Maîtrise des Systèmes et du Logiciel Conservatoire National des Arts et Mètiers, Paris, France, 3–5 December 2002. Paris, France: CMSL. Vol. 3. Session 11. 8 p.
Reprinted with permission from the publisher.

Quality-Driven Architecture Design Method

Mari Matinlassi and Eila Niemelä

VTT Electronics
P.O. Box 1100
90571 Oulu
Finland
Tel. +358 8 551 2111 Fax +358 8 551 2320

{Mari.Matinlassi, Eila.Niemela}@vtt.fi

Abstract: In this paper we introduce a quality-driven architecture design (QAD) method with three phases: system analysis, conceptual architecture design and design of a concrete architecture. Architecture design phases produce an aspect of software architecture at two different levels of abstraction. System analysis provides a bigger picture of the software system and its properties. Conceptual architecture organises these functional and quality responsibilities into conceptual elements, while a concrete architecture describes hierarchical components and communication protocols between them. Architectural descriptions at both abstraction levels are defined from three viewpoints: structural, behaviour and deployment. The QAD method has been applied within a case study of a distributed service platform with the mobility of system services.

Keywords: Software architecture, design methods, architectural descriptions, service architectures

1. INTRODUCTION

Architecture is the fundamental organisation of a software system embodied in its components, their relationships to each other and to the environment [2]. Software architecture also includes the principles guiding its design and evolution [10], and therefore, it has a strong influence over the life cycle of a system.

In the past, hardware engulfed other aspects of a system, and especially quality attributes like modifiability, interoperability and reusability were sacrificed first in the course of system development. Today, software-intensive systems are pervasive. Increasing complexity and size of software as well as the cost of software development and more mature software technologies have changed the role of software architecture and the importance of quality-oriented design.

An architecture design method should cover the following questions:

- What are the entities represented and manipulated by the design method?
- What are the concrete means to describe architecture?
- What are the design steps and their ordering?
- How to apply the method?

Software architecture has to be described so that various stakeholders, e.g. customers, management and developers, understand it. In this light, it is obvious that one kind of architectural description is not enough, but the architecture has to be described with several different views. Furthermore, different views support analysis of particular quality attributes [18].

The FORM method [13] presents a solution for transformation of requirements to an architecture style. However, software systems need more than one style to follow. Architecture styles and design patterns are applied as driving factors in [2] but mapping requirements to software architecture is quite vague.

Architecture Based Design (ABD) method [1] is a quality driven method for designing the software architecture for a long-lived system at the conceptual level of abstraction. In ABD, the conceptual architecture [9] is a representation of the high-level design choices described with three architectural views. In spite of the ABD method has been developed further to a new method called the Attribute Driven Design method, ADD [3] it still does not provide more than a coarse grained high-level, i.e. conceptual architecture as an output. Also the support for product line architecture design in the ABD and in the ADD is mentioned but immature [16].

UML, with the strength of earlier widely applied object oriented methods [12, 20, 22] gives it benefits as a unified modelling language [21]. However, its superiority as an architectural description language is debatable.

This paper represents a quality-driven architecture design (QAD) method [18] that provides a systematic way to transform functional and quality requirements to software architecture. In discussion, we will summarise our experiences on the use of the method for designing architecture of the case study and outline our future directions.

2. METHOD

When designing software architectures it is not feasible to begin with the bottom-up style because it expects one to concern the system in details. Instead one needs to use a top-down approach to the issue [4]. Conflicting practice of the architectural documentation today is that it does not support high-level architectural descriptions. With high level architectural descriptions available it is easier for adapters of the architecture to use a top down method when getting familiar with the structures and activities of a system. It is also improbable that an architectural design process would not require iterations to optimise an architecture. These reasons mentioned above cause a design method to be divided into conceptual and concrete phases with system analysis above all, as shown in *Figure 1.*

System analysis captures the technical properties and the context of the system. Conceptual architecture design phase models and documents the structure, behaviour and deployment of the system at an abstract level. Concrete architecture defines system structure, behaviour and deployment in a more concrete sense using architectural descriptions produced in the conceptual design. Quality of both architectures, conceptual and concrete, is assessed by the architectural quality analysis [6, 7] and thereafter, required changes are updated to architectural models.

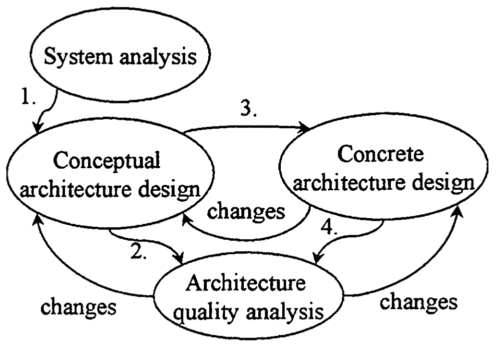

System analysis
1.
3.
Conceptual architecture design
Concrete architecture design
changes
4.
2.
Architecture quality analysis
changes
changes

Figure 1. QAD method's main phases.

A method should give a series of steps to follow [14]. The detailed steps of this method are described in [18] and are not included here because of space limitations. For the same reason, the architecture quality analysis is out of the scope of this paper. In the next sections are described the overview of the method and the main phases with related architectural descriptions.

2.1 System analysis

The purpose of system analysis is to link the requirements engineering phase of the development life cycle and the software architectural design. Requirements engineering considered here, identifies the driving ideas of the system and the technical properties on which the system is to be designed. Detailed functional requirements are to be clustered in the conceptual architecture design phase of the method.

2.2 Conceptual architecture design

The conceptual software architecture [1, 9] provides organisation of functionality and quality responsibilities into conceptual elements, collaboration between functional elements, and allocation of elements into hardware.

These different aspects of conceptual software architecture are represented with three architecture views: structural view, behaviour view and deployment view. A view is defined to be a representation of a whole system from the perspective of a related set of concern [10]. Every view produces its specific architectural descriptions (*Figure 2*).

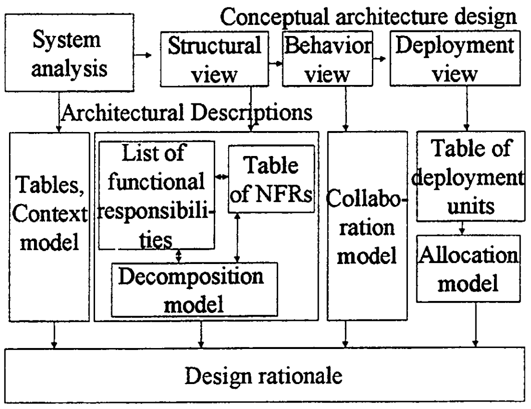


Figure 2. Design phases and architectural descriptions of conceptual architecture.

The first view describes the structural viewpoint: software elements that compose the system, their interfaces and interconnections. Hierarchical structure is illustrated in a logical model, which is built up by clustering functional responsibilities and mapped with the table of non-functional requirements (NFR).

The collaboration view specifies the behaviour of a system; dynamic actions of and within a system, the kinds of actions the system produces and participates in, as well as their ordering and synchronisation. The system behaviour is described with a collaboration model.

The third view, the deployment view, clusters conceptual components into deployment units and describes allocation of those units into physical computing devices. A table of units of deployment and a deployment model describe allowed allocations of units. The necessity of a unit in a system is presented in this view.

Design rationale is a set of design principles and rules. Design rationale also provides the reasoning why these principles and rules have been defined and possible consequences if they are neglected. Design rationale is related to an architectural description and can explain, for example, why a certain standard has been selected or the selected architectural styles with their preferences.

2.2.1 *Conceptual Structural View*

The structural architecture encloses conceptual subsystem components, leaf components and their conceptual relationships (*Table 1*). In addition, variability points specified at the conceptual level and architectural styles used are illustrated in diagrams.

Table 1. Conceptual structural design elements, types and responsibilities.

Element	Types of an element	Responsibilities of an element
Conceptual structural component	System component	What it has to do?
	Subsystem component	
	Leaf component	How it does it?
Conceptual structural relationship	Passes-data-to	
	Passes-control-to	Why? (Design Rationale)
	Uses	

Conceptual elements divide the system into large functional blocks. At the top level, the system is decomposed into subsystems and at the next level, every conceptual subsystem is divided into a few conceptual components. At the end of this decomposing process, conceptual components are the smallest blocks used in this conceptual architecture level. Partitioning into subsystems and conceptual components has to be done with respect to the functional and quality aspects. Quality attributes drive the selection of architectural styles and thus affect the forming of the system architecture.

Conceptual components have conceptual relationships between each other. Relationships are abstracted interfaces between components, describing if an element passes control or data to, or has uses dependency with another component.

The responsibilities of the element define its role within the system. Responsibilities include both functional requirements and quality-oriented items and should answer the questions 'how' and 'why' besides the simple, common question 'what'.

2.2.2 *Conceptual Behaviour View*

The conceptual behaviour view is used to specify the behaviour of a system at a high abstraction level. Behaviour is recorded by defining conceptual behaviour elements: components and relationships and the responsibilities these elements have in the system (*Table 2*).

Table 2. Conceptual behaviour design elements.

Element	Types of an element	Responsibilities of an element
Conceptual behaviour component	Service component	What it has to do?
Conceptual behaviour relationship	Ordered sequence of actions	How it does it?
		Why? (Design Rationale)

Conceptual behaviour components, i.e. service components, are derived from conceptual structural components. Service component is equal either to a subsystem component or to a leaf component defined in the conceptual structural view. Selection depends on the size and complexity of the system under development.

Service components have behaviour relationships between each other. Behaviour relationships are ordered sequences of actions among a set of service components.

Naturally, the number of action sequences in a complex system is infinite and thereafter only the most essential sequences of actions are considered here. Essential sequences of actions are called collaboration scenarios and each collaboration scenario has a set of services related to it.

The collaboration model is an aggregate of collaboration diagrams. Each collaboration diagram represents one collaboration scenario graphically. In earlier design steps the service components were identified and essential collaboration scenarios selected from an infinite number of action sequences. In this final step the question 'in what order are actions done in each scenario' is answered and drawn into a set of collaboration diagrams.

2.2.3 Conceptual Deployment View

The deployment view is used to identify the distribution of hardware nodes in the system, group conceptual components to units of deployment and specify possible allocation of deployment units in computing units (*Table 3*).

The system hardware is described by means of distributed computing units called deployment nodes. Each deployment node is a platform for various services. Combination of services in different deployment nodes may vary and thereafter, conceptual leaf components are clustered into units of deployment.

A deployment unit is composed of one or more conceptual leaf components. Clustering is done according to mutual requirement relationships between components. In other words, a unit of deployment is atomic in the deployment process i.e. it cannot be split and deployed on more than one node.

Table 3. Conceptual deployment design elements.

Element	Types of an element	Responsibilities of an element
Deployment node	Various, depends on system	What it has to do?
Unit of deployment	Mandatory, alternative, optional	How it does it?
Conceptual deployment relationship	Is-allocated-to	Why? (Design Rationale)

Each unit of deployment represents one of three alternative types. These types are mandatory, alternative and optional [13, 19].

Allocation relationships are recognised between deployment nodes and units of deployment. The allocation relationship represents which services will be deployed in which distributed nodes/devices.

2.3 Concrete Architecture Design

Concrete software architecture refines conceptual structural components and their relationships. Also collaboration between components is described in more detail and detailed lower level components are allocated to hardware.

These different aspects of concrete software architecture are represented with three similarly named architecture views as in the conceptual level of abstraction: structural view, behaviour view and deployment view. In this concrete approach to architecture, every view produces its architectural descriptions (*Figure 3*).

Architectural styles selected for the conceptual architecture are now complemented by design patterns [8]. These micro architectural elements [5] guide designers of components in greater detail to carry out the components and services compatible with the architecture. Appropriate design patterns are recorded as design rationale of the concrete architecture.

The first concrete view takes the decomposition model from the conceptual architecture as input and describes the structural viewpoint by a means of refining components and interfaces between them. Hierarchical structure is illustrated in structure diagrams, which are built up with respect to refined non-functional requirements.

The concrete behaviour view specifies the behaviour of each component. The concrete system behaviour is described with component state diagrams and message sequence charts.

The third concrete view creates software components that refer to concrete architectural components, i.e. to capsules [23] and also to protocols. Software component instances that can be allocated to hardware processors illustrate this kind of deployment in a system level diagram.

The QAD method has been experimented in a case study of distributed service platform (DiSeP). The purpose of the DiSeP is to make software components in a networked environment interact spontaneously. In the DiSeP, the software components are various kinds of services that are either a part of the platform or a part of the application that utilises the platform. Interaction denotes that distributed parts of the platform or an application provide services, and/or use services that are provided by somebody else.

The configuration of the network may change dynamically. The main goal is to maintain the interoperability despite the dynamic nature of the network and the differences in the hardware or in the software implementation.

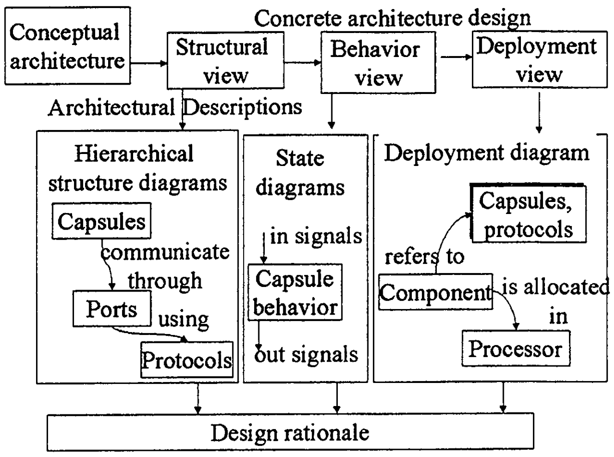


Figure 3. Design phases and architectural descriptions of concrete architecture.

The QAD method is also under validation in two research projects with industrial case studies. The other one is an international project of wireless internet service engineering and the other case considers integration of COTS components to a service platform.

3. DISCUSSIONS

We started by studying the ability of existing design methods to describe the characteristics of software architecture. The three methods, with four [9], 4+1 [15] and 3+1 [11] views, have differences in the naming of views, their descriptions and notations. In most cases, UML is used as a modelling language but also tables, natural languages and other modelling languages are appropriated. Therefore, we drew the conclusion that simplicity, consistence and understandability are the key issues of most importance in a method applicable for the development of architectures.

For simplicity, three viewpoints and two separate abstraction levels were defined. Three viewpoints were the minimum that was needed in the case study. However, development view for work allocation, third-party components and assets management is required in practice. Separation of abstraction levels makes a clear distinction between the issues described at the conceptual level and the matters considered in the concrete architecture development. In practice, the real problem is that the decisions that should be made at the component level are already defined in the conceptual architecture and therefore, architectural descriptions are confusing and difficult to understand.

Transformation of systems requirements to architecture and the use of the same viewpoints in the conceptual and concrete architecture assist in providing consistence between views. Traceability of systems properties is also possible due to mapping tables between the requirements and responsibilities of conceptual components. A recorded design rationale assists in understanding the comparisons, evaluations and decisions made during the development.

On the basis of the case study done with the QAD method, we see it as having several advantages. The method provides a systematic way to transform functional and quality requirements to software architecture and it also guides how to document the architecture. As a quality driven method, the QAD utilises architectural styles and patterns as a guide to carry out quality requirements in architectural descriptions. Quality-driven design provides that one defines the scope and requirement for each quality attribute involved in the system, because a quality attribute only gives an overall quality definition for the whole system [17].

Quality scope means encapsulating a requirement or requirements as a responsibility of a restricted part of the architecture. The restricted part is instantiated e.g. as a component or a group of interacting components. The quality requirement definition means a more exact and system specific description of the quality attribute in question. For instance, the quality requirements for distributed service platform [18] architecture (for availability attribute), may be: (1) Service 'A' must always be active (and serving) in one of the distributed units and (2) the instances of the Service 'A' must be activated and shut down under control.

Under the interpretation above, the scoping and definition of quality attributes are the essential activities in realizing quality attributes. Furthermore, they support the selection and localization of architectural styles and patterns. QAD supports the product line architectures through documentation of variability and it is especially aimed at service architectures, which are considerably raising their necessity. In addition to these, the QAD method provides systematic progression steps, it is simple to learn and applicable to existing modelling tools.

Despite the fact that the QAD method has several advantages, there are still several issues to improve. Because the method is new, it has to be validated with several industrial case studies. Case studies should cover different kinds of service architectures, e.g. wireless services, value-added services and ubiquitous computing. Thus, new viewpoints have to be defined, especially for the stakeholders defined in business models but also for the varied end-users of ubiquitous computing systems. The deployment view, especially in service architectures, seems to have an importance in assisting the organisation of the design work between software architects and component designers but also between subcontractors and the persons responsible for acquisition of third-party components. On the lower abstraction level the development view defines how conceptual components are realised by concrete source modules, executables, libraries and make files for assembling and configuring software systems by retrieving components from an assets repository. The concrete development view links the architectural views to the assets management.

Smooth linkage between the design and analysis of software architecture also needs further studies in order to provide a toolbox with a comprehensive set of analysis methods for all quality attributes. Further, the selection of architectural styles would be easier with a repository of styles supporting individual quality attributes. Design rationale is an equally important issue and the description of the design rationale must be unified through the design process.

Because QAD is the first version of the design method, the purpose of the method engineering was not to define an explicit method language. This is why experimental notations were used in addition to UML. However, strictly defined extensions for UML are needed and are under development.

4. CONCLUSION

In this paper, three viewpoints to architecture at two abstraction levels have been defined. Furthermore, suggestions on documenting these views with models and diagrams are introduced. To make these views and models useful, the method is experimented with upon existing tools in a service architecture case study and it is under evaluation in the WISE project[1] and in a national joint research project with industrial case studies.

To fulfil the quality requirements that have been set to a software product requires considering architectural viewpoints in the software development. Again, in order to reach quality attributes with architectural structures, the use of architectural styles and patterns are required. This is done through refining and scoping the quality attributes. Furthermore, software quality analysis in the early design phases supports quality-driven design.

In addition to quality requirements, software architecture has to answer to the functional demands defined by the customers and end-users. Utilising the constructs mentioned above, the QAD method provides an explicit and quality-driven link between software requirements and architecture.

Acknowledgements

Finally, we would like to acknowledge the support provided by the PLA program founded by VTT Electronics[2].

References

[1] F. Bachmann, L. Bass, G. Chastek, P. Donohoe, and F. Peruzzi: The Architecture Based Design Method; CMU/SEI, Technical report, 2000

[2] L. Bass, P. Clement, and R. Kazman: Software architecture in practice; Addison Wesley, 1998

[3] L. Bass, M. Klein, and F. Bachmann: Quality Attribute Primitives and the Attribute Driven Design Method; Proceedings of the Fourth International Product Family Engineering Workshop, PFE-4. pp. 163 – 176, 2001

[4] J. Bosch: Design & Use of Software Architectures; Addison Wesley, 2000

[5] F. Buschmann, R. Meunier, H. Rohnert, Sommerlad, and M. Stal: Pattern-oriented software architecture, a system of patterns; John Wiley & Sons, 1996

[1] A European project, IST-2000-30028.

[2] http://www.vtt.fi/ele/projects/pla/index.htm

[6] L. Dobrica and E. Niemelä: A Strategy for Analysing Product Line Software Architectures; VTT Publications 427, Technical Research Center of Finland, 2000, Available at URL: www.inf.vtt.fi/pdf

[7] L. Dobrica and E. Niemelä: Attribute-based product-line architecture development for embedded systems; Proceedings of AWSA'00, Monast University, pp. 76 – 88, 2000

[8] E. Gamma: Design patterns: elements of reusable object-oriented software; Addison-Wesley, 1994

[9] C. Hofmeister, R. Nord, R., and D. Soni: Applied Software Architecture; Addison-Wesley, 2000

[10] IEEE Std-1471-2000: IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems; IEEE Computer Society, 2000

[11] A. Jaaksi, A., J.-M. Aalto, A. Aalto, and K. Vättö: Tried & True Object Development. Industry-Proven Approaches with UML; Cambridge University Press, 1999

[12] I. Jacobson: Object-oriented software engineering: a use case driven approach; Harlow: Addison-Wesley, 1992

[13] K. C. Kang, S. Kim, J. Lee, and K. Kim: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures; Annals of Software Engineering, 5, pp. 143-168, 1998

[14] K. Kronlöf: Method Integration: Concepts and Case Studies; Chichester: John Wiley & Sons, 402 p., 1993

[15] P. B. Krutchen: The 4+1 View Model of Architecture; IEEE Software, 12, pp. 42-50, November 1995

[16] M. Matinlassi: Evaluation of Product Line Architecture Design Methods. ICSR2002, Young Researchers Workshop. Available at URL: http://www.info.uni-karlsruhe.de/~heuzer/ICSR-YRW2002/program.html

[17] M. Matinlassi, M. and E. Niemelä: Designing High Quality Architectures; International Conference on Software Engineering, ICSE2002, Workshop on Software Quality, 4 p., 2002

[18] M. Matinlassi, E. Niemelä, and L. Dobrica: Quality-Driven Architecture Design and Quality Analysis Method. A Revolutionary Initiation Approach to a Product Line Architecture; VTT Publications 456, Technical Research Centre of Finland, 2002. Available at URL: www.inf.vtt.fi/pdf

[19] E. Niemelä: A component framework of a distributed control systems family; VTT Publications 402, Technical Research Centre of Finland, 1999. Available at URL: www.inf.vtt.fi/pdf

[20] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen: Object-oriented modeling and design; Prentice Hall, 1991

[21] J. Rumbaugh, I. Jacobson, and G.Booch: The Unified modeling language reference manual; Addison-Wesley, 1999

[22] B. Selic, G. Gullekson, and P. Ward: Real-time object oriented modeling; John Wiley & Sons, 1994

[23] B. Selic and J. Rumbaugh: Using UML for Modeling Complex Real-Time Systems; Rational Software White paper, 1998. http://www.rational.com/products/whitepapers/100230.jsp.

PAPER V

The impact of maintainability on component-based software systems

In: Chroust, G. & Hofer, C. (eds.). Proceedings of the 29th EUROMICRO conference, New waves in system architecture. Belek-Antalya, Turkey, 1–6 September 2003. Los Alamitos, California: IEEE Computer Society. Pp. 25–32. ©2003 IEEE.
Reprinted with permission from the publisher.

The Impact of Maintainability on Component-based Software Systems

Matinlassi Mari, Niemelä Eila
VTT Technical Research Centre of Finland
Software Architectures Group
{Mari.Matinlassi, Eila.Niemela}@vtt.fi

Abstract

There is a great deal of inconsistency and vagueness in the treatment of and terminology involved with software maintainability. This is exacerbated by the fact that there are a number of different dimensions of maintainability, each requiring specific treatment. The trends of increasing systems functionality and increasing systems complexity have given rise to new dimensions of maintainability since ISO/IEC defined maintainability as "the capability of the software to be modified" in 1996. This paper introduces the framework of maintainability and the techniques that promote maintainability in three abstraction levels; system, architecture and component. In the system dimension, the maintainability requirement is considered from a business-related point of view. In architecture, maintainability means a set of quality attributes, e.g. extensibility and flexibility. At the component level, maintainability focuses on modifiability, integrability and testability.

1. Introduction

Software maintenance is long known as one of the most expensive and resource requiring phase of the software development process. Therefore, the requirement of maintainability and its impact on software development has to be clearly understood. The term "software component" has varying definitions. According to [1] software component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. In addition, a component conforms to and provides the physical realization of a set of interfaces. In addition, component-based software engineering (CBSE) is about developing, marketing and utilizing software components with their related assets. Therefore, CBSE goes well beyond enabling technologies e.g. JavaBeans and CORBA. CBSE includes wide-ranging issues from the theory of software reuse to the reality of commercial software markets, from available tools to programming language mechanisms and from practical testing to rigorous formal specification.

This study discusses maintainability, i.e. the capability of the software to be modified [2, 3] from the perspective of component-based software systems, wherein today, the concept of maintainability is of major importance [4], [5]. As a matter of fact, most of the software engineering methods and techniques e.g. reuse, product line approach and component-based software engineering have the same final goal: maintainability. For instance, reusability promotes maintainability through decreasing development costs. On the other hand, maintainability is a prerequisite for reusable software, because there is no meaning in e.g. a long-living reusable component that is not maintainable. As seen in many cases, some characteristics of maintainability can be seen as a pre-requisite for the provision of another.

The definitions for maintainability are many and its various nuances are often confused or misunderstood, as are all the other quality attributes [6], [7], [8]. Therefore, this study defines the dimensions of the maintainability requirement in component-based software systems and clarifies the impact of maintainability on software systems.

The requirement of maintainability permeates all levels of component-based software. Therefore, software developers need support for each maintainability dimension and in order to provide that support, it has to be understood what maintainability means and what its impact on component-based software systems is. In particular, maintainability of software that involves externally developed components differs from the traditional software maintenance in that the activity of maintenance is likely to be performed by someone other than the developer. This is the case whether the component is an 'in-house' developed component or a pure commercial component.

The structure of this paper is as follows. First, we define not only the pre-required quality attributes for maintainability but also techniques that promote maintainability. Then, we divide software into three abstractions; system, architecture and component dimensions and discuss the impact of quality attributes on each dimension. A case study of a product line of the traffic information systems illustrates the impacts of maintainability. In the end, we conclude our statements and draw out our future plans.

2. Background

2.1. Quality attributes

ISO/IEC Draft 9126-1 defines a software quality model [2] with six categories of quality characteristics: functionality, reliability, usability, efficiency, maintainability and portability. Quality characteristics are also called quality attributes [9], which can be categorized into execution and evolution quality attributes (also called simply 'qualities'). *Execution* qualities are observable at run-time. That is, they express themselves in the behavior of the system. *Evolution* qualities cannot be discerned at run-time, meaning that the solutions for evolution qualities lay in static structures of the software system. Therefore, they should be considered in the phases of the product's life cycle, i.e. in development and maintenance of a software system. Although we use categorization into execution and evolution qualities, other categorizations are available (see a collection e.g. in [10]).

Chung et al. [10] define a framework for representing the design process of non-functional requirements (or quality attributes, if you will). However, the framework does not categorize neither define the quality attributes explicitly but concentrates on recording the reasoning process behind the design decisions.

Table 1 and Table 2 define the execution and evolution qualities by extending the definitions in [11] with new attributes, namely being adaptability (execution quality) and extensibility (evolution quality). [3] denotes adaptability as a synonym for flexibility. However, we believe the meaning of adaptability and flexibility is different today and these attributes exist even in different attribute categories (execution and evolution). Adaptability means the ability of software to adapt its *functionality* according to the current environment or user, whereas the strict meaning of flexibility is about easy adaptation of software to environments *other than those for which it was specifically designed*.

Although the quality model includes functionality, i.e. the system's ability to do the work for which it was intended, we see it as a main category of execution quality attributes, realizing that functionality cannot be considered an *architectural* quality attribute [9]. However, interoperability, adaptability and reliability can be considered special, newly emerged forms of functionality, the forms that are architectural in nature [12]. While the characteristics of software systems are changing from monolithic to modular networked systems, and furthermore, to spontaneously self-organizing nets of adaptive computing units, new quality attributes are defined in order to characterize the qualitative properties of systems. Thus, the list of quality attributes is sensitive to changes in a similar way to attractiveness of systems' qualities.

Table 1. Execution qualities.

Attribute	Description
Performance	Responsiveness of the system, which means the time required to respond to stimuli (events) or the number of events processed in some interval of time.
Security	The system's ability to resist unauthorized attempts at usage and denial of service while still providing its service to legitimate users.
Availability	Availability measures the proportion of time the system is up and running.
Usability	The system's learnability, efficiency, memorability, error avoidance, error handling and satisfaction concerning the users' actions.
Scalability	The ease with which a system or component can be modified to fit the problem area.
Reliability	The ability of the system or component to keep operating over the time or to perform its required functions under stated conditions for a specified period of time.
Interoperability	The ability of a group of parts to exchange information and use the one exchanged.
Adaptability	The ability of software to adapt its functionality according to the current environment or user.

With a quick look, some of the evolution qualities in Table 2 seem to be almost the same. However, the attributes really have at least a different sound to their meaning. For example, maintainability may be equalized with modifiability [9]. However, we think maintainability is also affected by many other evolution quality attributes than modifiability. Modifiability includes adding, deleting and changing software structures and therefore, extensibility (also called expandability or extendability [3]) and portability can be considered special forms of modifiability. Furthermore, modifiability includes e.g. optimization and fault correction.

Two of the evolution qualities have a qualitative sound to their definition. In other words, two of the quality attributes define the others qualitatively as follows. Flexibility means the *ease* with which software can be modified and modifiability means the *quickness* and *cost-effectiveness* of modifications.

ISO/IEC defines sub characteristics of maintainability as follows: changeability, testability, analysability and stability. Changeability is a synonym for modifiability. The definition of testability in [2] is somewhat restricted and our definition of testability therefore covers both testability and analysability. As far as stability is concerned it equals to modularity [3] both meaning the degree to which software is composed of discrete components such that a change to one component has minimal impact on other components. These definitions are very close to coupling, i.e. *the manner and degree of interdependence between software modules* [3]. Low coupling is generally known as a basic rule of thumb in component-based software, not as a quality requirement or attribute of any particular system.

Table 2. Evolution qualities.

Attribute	Description
Maintainability	The ease with which a software system or component can be modified or adapt to a changed environment.
Flexibility	The ease with which a system or component can be modified for use in applications or an environment other than those for which it was specifically designed.
Modifiability	The ability to make changes quickly and cost-effectively.
Extensibility	The systems ability to acquire new components.
Portability	The ability of the system to run under different computing systems: hardware, software or combination of the two.
Reusability	The system's structure or some of its components can be reused again in future applications.
Integrability	The ability to make the separately developed components of the system work correctly together.
Testability	The ease with which software can be made to demonstrate its faults.

With the interpretation represented above, we conclude the definition of maintainability as in Figure 1. Maintainability is the ease with which a software system or component can be modified. Modifications may include extensions, porting to different computing systems or improvements. However extensibility and portability occur as distinct attributes, "improvability" does not – so far – appear in literature as a quality attribute. Improvements include correcting faults or exceeding any execution or evolution qualities of the system. Flexibility, reusability, testability and integrability contribute to modifiability and therefore, are defined as sub attributes of maintainability.

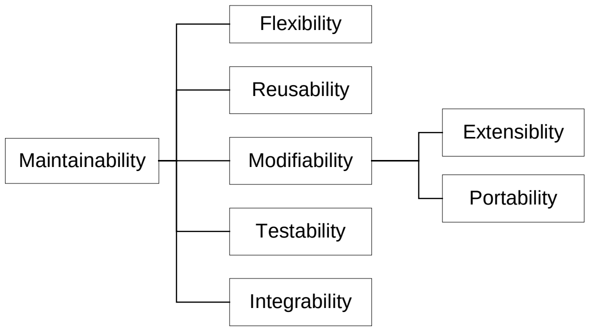


Figure 1. Sub attributes of maintainability.

2.2. Other characteristics related to maintainability

In this section, we introduce other characteristics related to maintainability. Often, these characteristics are confused with quality attributes, but actually they are *techniques* that promote and support achievement of maintainability and its sub-attributes.

2.2.1 Traceability. Traceability is the ability to document and follow the life of a concept throughout system development. It is forward directed (post traceability: describing the deployment and use of a concept) as well as backward directed (pre traceability: describing the origin and evolution of a concept) [13]. Although traceability is an essential characteristic of the component-based software development to achieve a maintainable solution, it is a *supporting technique* to achieve the qualitative property of the artifacts produced during the development process.

2.2.2 Variability. According to [14], the differences among products are managed by delaying design decisions, thereby introducing variation points, which again are bound to a particular variant or variants. A variation point identifies a location at which a variation can occur in the system [15]. Therefore, variability is not a quality factor as such, but it provides a mechanism to manage the anticipated changes in software structure(s) during the evolution of systems, thus increasing all sub-qualities of maintainability**.**

Different (and overlapping) types of variants are introduced [16], [17], [18], but the most commonly used are:

- Mandatory; the type included in all products in the domain.
- Optional; the type for any product in the domain.
- Alternative; a choice that cannot coexist with other alternatives (in the same variation point).

Possible dependencies between variants are the 'requires' relationship and the 'mutually exclusive' i.e. mutex relationships.

2.2.3 Tailorability. Tailorability is a loose term used in component-based software development to describe the ability to customize and configure components, but also to add new components to the system and combining services of multiple components in novel ways [4]. Here we consider tailoring as a technique (like variation points) but instead of software architecture, tailorability focuses on customizing the *internal capabilities* of components according to customers' needs.

2.2.4 Monitorability. Isolating the faults in a component-based system is difficult, especially in systems that utilize third party components, because the integrator has to ascertain (1) how the components work and (2) why they do NOT work. The source of the difficulty is obvious: the integrator has no visibility into the components and no control of their operation [19]. Therefore, monitorability, that is the systems property to support e.g. measurement of performance and resource usage, watching for failures, chase up security violations or monitoring of user behavior, is an essential property for a maintainable system [4].

According to [19], the classification of monitoring capabilities is as follows:

- Intra-component; these techniques observe the behavior of a component, to understand and demystify the component developer's assumptions and intended usage of the component.
- Inter-component; these techniques observe the behavior of two or more components, to understand potential mismatches between components.
- Extra-component; these techniques observe a system of cooperating components, to understand macro-level issues dealing with performance and misfits, etc. That is to say, extra-component monitoring is system level monitoring.

A Built-In-Test (BIT) component [20] is a component model that has one or more test interfaces (for intra-component monitoring) and a test mechanism (for inter-component monitoring) embedded in a component. BITs offer two benefits: the component user can check the behavior of the component and the component can check if it is deployed in an inappropriate environment. Thus, monitorability is a technique that promotes testability.

3. The impact of maintainability

Maintainability has an impact on three abstraction levels: system, architecture and component dimension. In the following, we define the impact of the maintainability requirement on all dimensions. Table 3 summarizes the impact of maintainability on a software system (S), architecture (A) and component (C). 'X' means quality attribute having a certain impact at the level in question. Each column in Table 3 and Table 4 will be concerned in more detail in the following sections from 3.1 to 3.3, wherein the differences of attribute impacts at each level are discussed.

Table 3. Quality impacts on the dimensions of maintainability.

Attribute	S	A	C
Flexibility		X	
Reusability	X	X	X
Modifiability		X	X
Extensibility	X	X	
Portability		X	
Testability	X	X	X
Integrability		X	X

Table 4 illustrates how each technique related to maintainability promotes the sub-qualities of maintainability. Also, the dimensions of impact (S, A or C) are defined. Tailorability largely means modifiability in the component dimension. Monitorability is expressed in all dimensions as follows. Extra-component monitorability affects system level testability and second, inter-component monitorability affects testability on the architectural level and finally, intra-component monitorability is conducive to individual component testability. Variability appears largely in the architecture dimension but also in the system dimension whereas traceability must be provided from the system level through the architecture and the design to the components.

Table 4. Factors conducive to quality attributes.

Technique	S	A	C	Attribute
Tailorability			X	Modifiability
Monitorability	X	X	X	Testability
Variability	X	X		Extensibility
Traceability	X	X	X	Maintainability

These key factors may be similar to what normally exist also in non component-based systems. However, using components means that the nature of these maintenance activities changes as described below.

3.1. System dimension

When the overall quality requirement is maintainability, at the system level its value is considered from business-related points of view as follows [21]:

- Estimation of the effort required for adopting software to other contexts, i.e. for producing other products, sub-systems or components. How often is this kind of work required?
- Extent of software usable for future products. When will it be utilized?
- Identification of the execution platforms upon which software should be executed. Why are they selected?
- Identification of software that might be changed during the life cycle of the product. Which standards will be adhered to?
- Assumptions of the extended purpose of the system. Why and how is it happening?
- Anticipation of maintenance cost based on estimated length of the life cycle of the system. What does an upgrade cost, and how often is it required?

Discovering the facts of the above-mentioned list of issues attempts to assist in finding out the type, scope and position where realization of maintainability provides the most benefits. After prioritizing the sub-qualities of maintenance, the appropriateness of supporting techniques has to be estimated.

Monitoring of behavior and system resource usage gives information required when changed systems have to be tested. Monitoring techniques also set prerequisites for the system-level test support.

On the other hand, *variability* management provides a mechanism to handle the hot spots of changing functionality, structure, behavior and allocation.

Post *traceability* of requirements through architecture and components to code requires robust and reasonable documentation, but it also provides a path to put maintainability into practice and a tracking mechanism for quality maintenance.

A standard way of providing traceability is the establishment of cross-reference data. Such references can be expressed as links or matrices where connections between the various artifacts in code, architecture and requirements are made explicit [17].

3.2. Architecture dimension

In the architecture dimension maintenance definitely means *modifications* that have to be done quickly and cost-effectively. Modifications may include *porting* the system into a new operating system or other environment, or *extending* the system with new functional features. That is, quality requirements (derived from maintainability) that should be considered at the architectural level are at least *modifiability*, *portability* and *extensibility*.

Extensions to the architecture may also mean the integration of third party software components. Therefore, integrability concerning future third-party components also has to be supported in architecture.

In spite of the fact that architecture has been designed to be easily modifiable for changes that can be predicted, in order to have a maintainable architecture, the modifications that are not predicted should also be easy to do. Easy modifications for environments the architecture was not initially designed for means *flexibility* of the architecture.

So far, all the maintainability impacts described above have been related to modifying the architecture. Testing the modified parts of the architecture follows modifications. As a matter of fact, *testability* is the capability of the architecture to enable modified software to be validated. However, at the architecture level, testability means quality analysis, i.e. evaluation of architecture and how maintainable it is. There are several appropriate scenario-based analysis methods that can be applied [22]. However, in order to be testable the architecture has to be documented properly [23].

Reuse of architecture may be the only way to implement reusability when implementation technology is changing from one product to another. Although it is not the case in most component-based software systems, architecture sets the conditions, scope and time when reuse is possible and beneficial.

3.3. Component dimension

The impact of maintainability varies depending on the size of the component. At least the following maintainability impacts apply to small components:

- Integrability, i.e. conformance to component model and standards used in the system the component is to be integrated in.
- Interoperability with components from many different vendors. Although interoperability is an execution quality, it is related to integrability and transitively to maintainability.
- Modifiability, how easy it is to modify the component to satisfy local requirements.
- Testability of black box components through monitoring intra-component behavior and failures.

In the case of large software components that have their own architecture or perhaps a product line architecture, the impacts are naturally the same as the impact on architecture dimension as described above.

Monitoring black-box components is difficult because the intra-component behavior is hidden. Thus, rather than intra-component capabilities maintainers must use inter-component and extra-component monitoring capabilities to observe system behavior [4]. Component suppliers are responsible for intra-component monitoring capabilities through creation of special open monitoring interfaces.

Business factors also affect maintainability of commercial components [4], [24]. The following criteria [24] aim to address the factor of system evolution in component selection:

- Vendor business stability. how long has the vendor been in business? Is there a risk of the vendor going out of business?
- Development process. What kind of testing process does the vendor use? Is the certification process at the vendor site appropriate?
- Obsolescence of the component. What happens if the vendor goes out of business? What happens if the component becomes obsolete?
- Maintenance contract. Who (vendor/integrator) is responsible for the component maintenance and to what degree?
- Stability of the component. What does the component version history reveal? How high is the frequency of upgrades? What are the reasons for upgrades?
- Marketing trends. What are the technology alternatives on the markets? Are there alternative, comparable components available in the market?
- Availability of customer support. How complete is the customer support for the component? What is the form of the support (phone-based, online, discussion groups etc.)? Is the support cost in balance with the assistance provided by the vendor?

4. Case study

4.1. Overview

Our case study, a product line (PL), consisted of different kinds of client terminals used for fare collection in public transportation. When considering a product line we define the first two product line members as:

- P1; a driver terminal, used for fare collection, travel card loading and usage, data transfer and locating. The device is a fixed-point device located in a vehicle or point of sales.
- P2; a conductor terminal used for fare collection, travel card loading and usage, data transfer and locating. The device is a wireless, portable terminal used in a vehicle.

At the end of the day, accountings are transferred from the Px devices to an office system at the depot. The office system states for software connected with the PC computers of the X system. However, the office system's software was out of the scope of this product line.

Both products share a common software architecture and many common features. Both product line members also have in-product variability, meaning different product versions depending on the target country and market area.

The aim of this case study was to improve the maintainability of the software product line. Maintainability requirements were discovered in a product line phase, wherein one product (P1) was already implemented and the first customer deliveries of that product were just about to emerge (Figure 1). The feasibility study of the second product, the family member P2 was ongoing.

The case study *concentrated* on considering maintainability from the perspective of architecture. In addition, the case study included one OCM (Original software Component Manufacturer) and one COTS (Commercial-Off-the-Shelf) component, which refined the scope of the example. The OCM component serves for data transfer between the product (P1 or P2) and the depot, whereas a database was acquired as a COTS component. The system level dimension of maintainability was achieved through considering the system from the perspective of overall functional and quality requirements of software. However, marketing trends, emerging standards and future advances in hardware capabilities also affected the maintainability requirements in the system dimension.

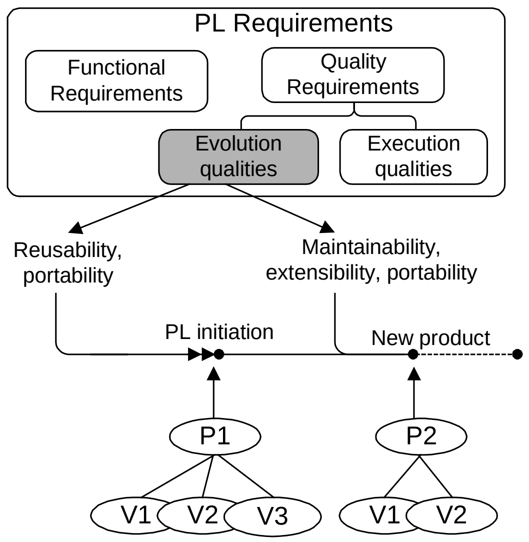


Figure 1. Case study: extension of a PL.

4.2. The impact of maintainability on the case study

Table 5 summarizes the impact of maintainability requirements on the PL case study with two product members. In the table, lines with gray shading concerned the case study. The main quality requirement was maintainability. During the evaluation, it revealed that in this particular case, maintainability was expressed in the form of modifiability, extensibility and portability.

Table 5. Maintainability dimensions in the case study.

Attribute	S	A	C
Flexibility		X	
Reusability	X	X	X
Modifiability		X	X
Extensibility	X	X	
Portability		X	
Testability	X	X	X
Integrability		X	X

At the architecture level, modifiability included requirements such as deleting features that are related to obsolescent travelling card technologies. At the component level, the OCM component for data transfer was required to be easily customized (that is, component level modifiability) to support different data transferring technologies.

At the system level, extensibility included the possibility to easily extend the product features towards public transportation information management, instead of bare fair collection and accounting. Thus, at the architecture level, extensions to the architecture were implemented through variation points. Variability was traced through the chain of development artifacts.

Portability in the architecture dimension meant capturing the environment-specific software into components or layers that encapsulated environment users from the environment. Environment here means e.g. different peripherals, display and printer types. Therefore, in some of the portability requirements variability was used as a supporting technique. One of the portability requirements was also that the software should be portable to a single fixed node and to a distributed environment with the main module and a trip computer.

5. Conclusion and future work

Maintainability permeates all levels of component-based software and therefore its nature is often misunderstood. In order to clarify the meanings of maintainability, its relations to other qualities were analyzed resulting in a framework that attempts to assist software developers become aware of when, where and how they should pay attention to the many faces of maintainability.

Maintainability concerns the whole life cycle of the component-based software, and therefore, it exists at all abstraction levels in software development. We identified three levels: system, architecture and component. In these dimensions, maintainability means different things, and therefore, techniques to achieve it also vary. However, traceability from one level to another is the key; if it is omitted, investments will not be returned. That is why our further work will focus on the qualities essential for evolvable systems and methods and techniques to develop and keep them living on.

Acknowledgment

We would like to thank Buscom Oy for providing the case study. The publication of this paper has been supported by a national joint research project funded by the National Technology Agency (Tekes), VTT and Finnish industry.

References

[1] A. Brown and K. Wallnau, "The Current State of CBSE," *IEEE Software*, vol. September/October, pp. 37 - 46, 1998.

[2] ISO/IEC, "Information Technology - Software Quality characteristics and metrics - Part 1: Quality characteristics and sub-characteristics,"., 1996, pp. 21.

[3] IEEE, "IEEE standard glossary of software engineering terminology," in *Std 610.12-1990*: IEEE, 1990, 84p.

[4] M. Vidger, *The Evolution, Maintenance and Management of Component-based Systems*. Boston: Addison-Wesley, 2001.

[5] P. Bengtsson and J. Bosch, "Architecture Level Prediction of Software Maintenance," in *Proceedings of the Third European Conference on Software Maintenance and Re-engineering*, 1999, pp. 139 - 147.

[6] O. Preiss, A. Wegmann, and J. Wong, "On Quality Attribute Based Software Engineering," in *The Proceedings of the 27th Euromicro Conference*: IEEE, 2001, pp. 114 - 120.

[7] M. Bertoa and A. Vallecillo, "Quality Attributes for COTS Components," presented at ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Malaga, Spain, 2002.

[8] J. Offutt, "Quality Attributes of Web Software Applications," *IEEE Software*, vol. 19, pp. 25 - 32, 2002.

[9] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Reading, Massachusetts: Addison-Wesley, 1998.

[10] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos, *Non-functional requirements in software engineering*. Boston, Dordrecht: Kluwer Academic Publishers, 2000.

[11] L. Dobrica and E. Niemelä, *A strategy for Analyzing Product Line Software Architectures*, vol. 427. Espoo: VTT Electronics, 2000.

[12] L. Davies, Gamble, R. F., Payton, J., "The impact of component architectures on interoperability," *The Journal of Systems and Software*, vol. 61, pp. 31-45, 2002.

[13] M. Anastasopoulos, J. Bayer, O. Flege, and C. Gacek, "A Process for Product Line Architecture Creation and Evaluation, PuLSE-DSSA," IESE, IESE-Report 038.00/E, June 2000 2000.

[14] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, and K. Pohl, "Variability Issues in Software Product Lines," in *Proceedings of the 4th International Workshop on Product Family Engineering, PFE-4*. Bilbao, Spain: European Software Institute (ESI), 2001, pp. 11 - 19.

[15] S. Salicki and N. Farcet, "Expression and usage of the Variability in the Software Product Lines," in *Proceedings of the 4th International Workshop on Product Family Engineering*. Bilbao, Spain: European Software Institute (ESI), 2001, pp. 287 - 297.

[16] K. C. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis. Feasibility study,," Software Engineering Institute, Pittsburgh CMU/SEI-90-TR-21, 1990.

[17] M. Anastasopoulos and C. Gacek, "Implementing Product Line Variabilities," in *Proceedings of the SSR'01*: ACM, 2001, pp. 109 - 117.

[18] F. Bachmann and L. Bass, "Managing Variability in Software Architectures," in *Proceedings of the SSR'01*: ACM, 2001, pp. 126 - 132.

[19] S. Hissam, "Experience Report: Correcting System Failure in a COTS Information System," in *Proceedings of the International Conference on Software Maintenance*: IEEE, 1998, pp. 170 - 176.

[20] H. Edler, Hörnstein, J., "BIT in software components," EC 5th Framework Project IST-1999-20162. 2001.

[21] E. Niemelä and T. Ihme, "Product Line Software Engineering of Embedded Systems," in *Proceedings of SSR'01, Symposium on Software Reusability*. Toronto, Ontario, CA, 2001, pp. 118 - 125.

[22] L. Dobrica, Niemelä, E., "A Survey on Software Architecture Analysis Methods," *IEEE Transactions on Software Engineering*, vol. 28, pp. 638-653, 2002.

[23] A. Purhonen, E. Niemelä, and M. Matinlassi, "Viewpoints of DSP Software and Service Architectures," To be appeared in the *Journal of Systems and Software*, 17p, 2003.

[24] S. Yacoub, A. Mili, C. Kaveri, and M. Dehlin, "Hierarchy of COTS Certification Criteria," in *Proceedings of the First Software Product Lines Conference*, P. Donohoe, Ed. Boston: Kluwer Academic Publishers, 2000, pp. 397 - 411.

PAPER VI

Architecture-centric approach to wireless service engineering

In: Annual Review of Communications 2003.
Vol. 56, pp. 875–889.
Reprinted with permission from the publisher.

Architecture-Centric Approach to Wireless Service Engineering

Eila Niemelä, Ph.D.
Research Professor
VTT Technical Research Centre of Finland, Oulu, Finland

Mari Matinlassi, M.Sc.
Research Scientist
VTT Technical Research Centre of Finland, Oulu, Finland

Patricia Lago, Ph.D.
Assistant Professor
University of Turin, Politecnico di Torino, Italy

Abstract[1]

Telecom carriers, wireless application service providers, and traditional Internet service providers (ISPs) are developing new services and new business models to support the mobile customer and create new revenue opportunities. At the same time technologies are evolving faster and faster and providing new features that make software engineering both promising and challenging for this domain. Next generation networks (NGNs) and services, GRID services and mobile services over 3G and 4G technologies, such as I-mode and Universal Mobile Telecommunications System (UMTS), represent some examples.

In this evolving scenario industry requires software engineering techniques that help in mastering time-to-market service engineering, fast and profitable evolution, and know-how protection and exploitation. In order to respond to the needs of various stakeholders related to service architectures, architecture descriptions must contain several viewpoints, at different levels of abstraction, defined by the quality-driven architecture design and quality analysis (QADA) method. To achieve this multiperspective representation, differing modeling notations for both abstract and concrete architecture descriptions are needed. This is to prevent confusion caused by diverse meanings for the same symbol. In particular this paper proposes a service-engineering approach for architecting wireless services. The approach relies on a modeling notation extending Object Management Group (OMG) unified modeling language (UML), and it is based on two separate levels of abstraction:

- High-level notation should enable drafting and understanding the whole of a system. It means that conceptual models should be easy to modify and should not contain too specific details. It should also provide a suitable communication mean among stakeholders that need to interact on a technical basis but also consider business issues.
- Low-level notation should support detailed design. It means that concrete models should integrate the neglected or informally described details of high-level models. It should also support design-level reuse by providing both context-independent and context-dependent models.

In order to facilitate understanding, service engineering requires readable, simple, and intuitive notations for both the conceptual and concrete architecture descriptions. High-level notation at the conceptual level should allow the grouping of functionality of services according to commonalties and variables and assist in the creation of interdependencies between the services. It also provides the means to draft service and work allocation for a distributed system in a distributed development environment. On the other hand, the notation at the concrete level should allow the separation of the externally and internally visible structure and behavior. In addition distributed interfaces, local interfaces and interactions among components and with external products should be clearly identified. The latter needs special attention as interactions typically take place among different business entities, using different protocols, standards, and business policies.

1. Introduction

The general software architecture of a future wireless telecommunication system can be divided into system infrastructure services, middleware, and applications. Service architecture is the architecture of applications and middleware. Infrastructure services are based on access technologies, digital signal processing (DSP), software, and network services. There are at least three reasons why the role of software architecture has changed. First the architecture of middleware services and applications is based on the widely accepted assumption and consensus that the wireless and mobile access systems will be converged with Internet systems. Nowadays infrastructure services form the largest category of software products available on the market, but the maturing of software service solutions is going to extend the global software market for generic middleware services. Therefore the quality of software services will become a vital factor, especially for service providers and service developers who purchase and rent software from third parties and extend the use of open source software in their platforms.

Second the increased size and complexity of software systems has also led to a need for more explicit definitions and descriptions of architecture. Architecture is the structure or structures of a computing or software system, which are comprised of software components and the externally visible properties of those components and the relationships among them (Bass, 1998). This architecture also must meet the functional and quality requirements set by different stakeholders. For service providers the service platform is a long-term investment that must be used in a set of products in order to be cost effective, but end users prefer real added value at a reasonable price. Service architecture should consider and balance the quality requirements before the development and during the evolution of a service.

Third the domain of wireless Internet technologies is on the leading edge of technological development. This means that industrial companies in the wireless service domain are pioneers, i.e., as early as new technologies are available, they are eager to apply them in order to develop new richer and more attractive services for their customers. From the technological point of view, this means that reusable design knowledge should be presented in an implementation-indent way and in a form that can be adapted to several kinds of execution environments. The more requirements, the more complex is the architecting, and that is why architecting guidelines must be developed, i.e., principles for how to develop and maintain the software architecture of wireless services.

The taxonomy of the formally defined orthogonal properties of software architectures (TOPSA) (Bratthall, 1998) extends the definition of software architecture defined earlier in Bass' work. TOPSA defines a space with three dimensions: abstraction (conceptual or realization); dynamism (static or dynamic); and aggregation. Accordingly, our contribution is the QADA method that defines a conceptual architecture description for identifying software architecture in terms of abstract criteria (Matinlassi, 2002), whereas a concrete architecture description captures architectural issues closer to software realization. In addition both architectural descriptions need several viewpoints in order to represent the whole system from various perspectives. Every viewpoint of conceptual architecture is an abstraction of the ones in concrete architecture. Abstraction means the selected removal of information, i.e., bigger components, fewer details, and deferred functions. Conceptual architecture descriptions are essential in the early phases of design when roughing out the structures of software in order to reach a common understanding with the technical and nontechnical stakeholders involved in the development of a software service or a system. For example managers analyzing which service level agreements (SLAs) influence the deployment of a certain service category and how demand a business view. This is not suitable for analysts who study which functional features need to be developed from scratch, rather than reused or bought. In these cases a view of functional requirements or usage scenarios is more appropriate, e.g., a use-case view. Again business and use-case views are not suitable for developers that need to map software components on networked devices or machines to agree on communication protocols, exported interfaces, security policies and information exchange.

Conceptual and concrete architecture descriptions are to be made with notation for which we define requirements and practical reasons. First the architectural descriptions represented with the notation should be expressive and intuitively understood. Second notation should be simple, i.e., easy to learn and use. In addition the notation should conform to the UML standard. Third it should support separation of concerns, i.e., orthogonal properties of software architectures are visible and manageable. Fourth the notation should assist the communication between the stakeholders involved in architecting. Finally the architectural descriptions should also be easily maintainable, reusable in different contexts, and still specific to the implementation decisions chosen for the current development.

Here architecture modeling is considered the domain of wireless services in particular. Service architecture is a set of concepts and principles for the specification, design, implementation, and management of software services (TINA). A service is the capability of an entity, such as the server, to perform, upon the request of another entity, in this case, the client, an act that can be perceived and exploited by the client. This paper introduces the reasoning and background for the two levels and four viewpoints in service architecture modeling and especially how these viewpoints are intended for the use of technical and business stakeholders, such as vendors, operators, and service providers in a multiorganization development environment. Furthermore it introduces the notation of architecture modeling based on UML at both levels of abstraction and exemplifies the wireless service engineering (WISE) approach by examples of a game service that is under development in the WISE project.

The paper is organized as follows: after an introduction to the problem of engineering wireless service architectures, the second section identifies the requirements from the perspective of different stakeholders. Requirements are mapped on the two abstraction levels introduced in the preceding paragraphs. The third and fourth sections focus on the architectural descriptions of the service engineering approach for both conceptual and concrete levels respectively. The fifth section reports initial experiences gained in applying the approach to pilot projects developing wireless services, e.g., trading on-line through mobile terminals and entertainment applications such as interactive gaming. Conclusions and directions for future work close the paper.

2. Development of Wireless Service Architectures

2.1. Stakeholders of Service Architecture

When considering the use of service architectures, we found several reasons why the architecture of wireless services is a fundamental tool in communication and cooperation for the persons involved in the development of a service. The service architecture is used for communication in order to

- Get an overview of available services and their use,
- Classify needed services into generic and specific categories,
- Describe responsibilities and context of services and components,
- Consider the appropriateness of service architecture (technical and business issues),
- Prioritize quality attributes of the service architecture and reasoning them,
- Evaluate how quality requirements are achieved with architectural styles and patterns and
- Understand and integrate third-party components used in the service development.

To achieve better cooperation between team members, the service architecture is essential in

- Allocating and understanding the work division,
- Mapping services to components and vice versa,
- Mapping functional and quality requirements to services, and
- Clustering the components to be developed into potential technology domains.

As demonstrated the intention of software architecture is communicative; it is developed and used mainly for gaining a better understanding of what to do and sharing this understanding. That is why software architecture must be described in several ways, i.e., to present a slice of architecture in a certain light so that various stakeholders, such as customers, marketing and production staff, technical and administrative managers in addition to the software and hardware developers, understand it. A common understanding is difficult to achieve, because even if the basic intention of stakeholders is similar, different stakeholders need information at different levels of abstraction and aggregation.

Table 1 summarizes the stakeholders identified to play a certain role in wireless service engineering. In service engineering the main interest of service developers, service providers, and content providers are the services to be offered to the end users. A system architect develops a system structure that meets the requirements and constraints set by the earlier mentioned stakeholders. A software architect or a software product line architect plays the same kind of role in software development. He or she is responsible, however, for showing that the defined requirements are also met on the software architectural level. That is why it is obvious that one kind of architectural description is not enough, but that the architecture must be described with several different views.

Stakeholders of service development are closely associated with the roles of wireless service business (see *Figure 1*). Service users take advantage of deployed services, and service providers market services to customers. In addition relatively new business roles are content providers whose business is to sell providers the information needed to operate a service, e.g., movies, literature, and social statistics, and network operators who sell to service providers network capabilities needed to execute end-user services. Differing from

TABLE 1

Service Architecture Engineering Stakeholders

Category	Stakeholder	Description
Services	System architect	Develops a system architecture, HW and SW partitioning
	Service user	Uses services defined by the service architecture
	Service provider	Provides services for service users
	Service developer	Develops services for service providers
	Content provider	Offers content for service providers
Components	Component designer	Designs components that provide services
	Component integrator	Integrates available components into services
	Component developer	Designs, implements, and tests software components
Products	Product architect	Creates a product architecture
	Product developer	Develops product specific part of software, integrates components
	Product marketing	Presenting product (variable) features to customers
Software	Manager, assets manager, reuse manager	Manages, deals with costs and benefits, business, technology and reuse strategies
	Software architect	Develops software product (line) architecture
	Testing engineer	Tests software packages, integration testing
	Maintainer	Upgrades products and systems

other roles application and technology providers do not play an active role in service provisioning. Instead they produce applications, such as graphical metaphors for user interfaces, and technologies, such as mobile devices, sold to service providers to make up services, and after service deployment they are no longer involved in wireless service business.

2.2. Quality of Service Architecture

The main roles of stakeholders, i.e., a service user, a service developer, and a service provider, were used as a starting point to identify the essential quality attributes of wireless service architecture.
Figure 2 represents an overview of the quality stack that classifies qualities into internal and external qualities of four categories. Here we consider only the three upper levels that match the scope of service architecture.

The internal qualities are the nonfunctional properties of software service that are important for the developers of that part of the software in question but may be invisible or unimportant to the other stakeholders involved in the service development. The external qualities are the quality requirements that have to be visible to the stakeholders that use the software when they develop or provision the final product, a software service.

Various stakeholders in wireless services, i.e., users, application developers, platform or middleware service developers, and network operators, prefer different qualities. External quality provided by a stakeholder is a prerequisite for internal quality of another stakeholder in the stakeholder stack. The real quality of a service, or how well the service meets all end-user's requirement, weighing the cost versus benefits, defines the real added value for an end user. This quality is achieved only if prerequisite technical and economic qualities are met.

Applicability, or how easily the application can be applied in different contexts, is a quality that the application developers are most interested in. This quality is visible as external qualities through a graphical user interface's (GUI's) usability, performance of the application, and ease of service use by a scaling number of end users.

Interoperability of platform services is the criterion a service developer considers a required quality of the software when a service is provisioned. Interoperability is achieved if platform services such as middleware with communication and management services are generic and new platform services can be easily integrated by aggregating the old ones, which is horizontal integration. The same platform services should also be usable in new sets of applications, which is vertical integration, and application developers should be able to use them easily enabling simplicity of provided application interfaces. In order to be profitable with regard to development cost and time-to-market from the service provider's point of view, the platform services should also be portable, modifiable for different applications, expandable, maintainable, and easily used and accessed by application developers.

In summary the following quality attributes are the most important in the development of service architectures:

- External qualities: integratability, reusability, and simplicity
- Internal qualities: portability, modifiability, performance, and usability

Some of these qualities can be analyzed after the first implementation is ready, i.e., with the use of the platform services such as performance, but some are visible in the service architecture even from the first draft, namely portability, maintainability, integratability, and simplicity. We will now turn to the different viewpoints required in service architecture modeling.

2.3. Viewpoints of Service Architecture

An architectural view is a representation of a whole system from the perspective of a related set of concerns (IEEE, 2000). In the literature there are several approaches to the design of software architecture that concentrate on different

FIGURE 1

Roles in the Wireless Services Business

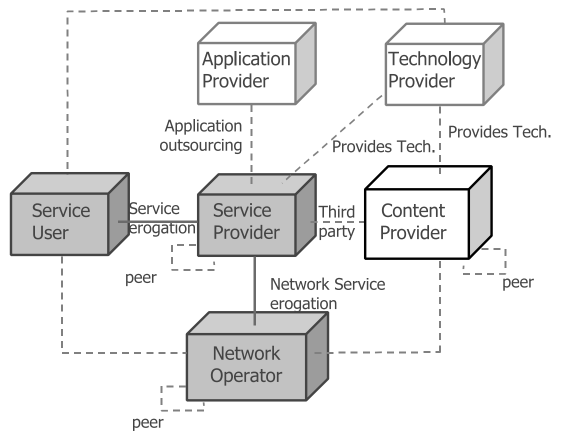
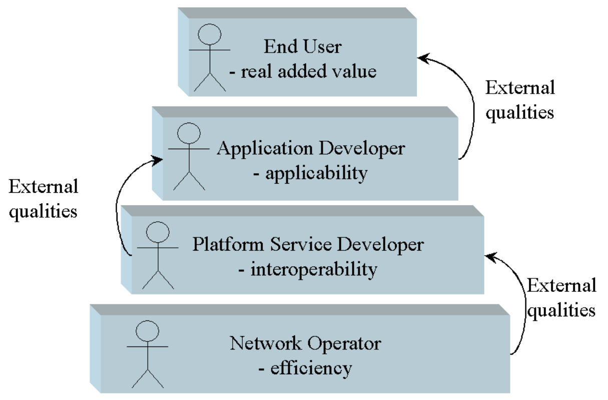


FIGURE 2

The Real Quality Accumulates in Cooperation with Various Stakeholders



views of architecture. The first of these view-oriented design approaches was the 4+1 approach (Krutchen, 1995). After this several others have approached the jungle of architectural viewpoints (Hofmeister, 2000; Jaaksi, 1999). Among these approaches there is no agreement on a common set of views or on ways to describe the architectural documentation. This disagreement arises from the fact that the need for different architectural views and architectural documents is dependent on two issues: system size and the domain, e.g., the wireless services domain. Again both the system size and domain have an impact on the amount of different stakeholders. Therefore it is obvious that none of these methods alone is comprehensive enough to cover the design of software architectures for systems of a different size in various domains or provide an explicit means to create architectural descriptions for all the systems.

Here we concentrate on the service architecture domain and the viewpoints needed in service architecture modeling. The definition of viewpoints is based on the three viewpoint elements defined in the QADA method extended with a definition of the fourth viewpoint, the development viewpoint (Matinlassi, 2002 A and B). Viewpoints for both levels of abstraction, conceptual and concrete, are similarly named: structural, behavior, deployment, and development (see *Figure 3*). These viewpoints embody the quality of service architecture and a service developed by using it. Qualities are visible at the architectural level only through the documentation of service architecture, views, models and diagrams, and notation used in them, as well as reasons behind the design decisions or the design rationale.

Fragments of the viewpoint elements are shown in *Table 2* and *Table 3* (Purhonen, in review). The tables capture the issues each view concerns. These issues are aimed at certain stakeholders. Each view also produces its own specific artifacts such as models or diagrams that provide appropriate information for the stakeholders. The differences between the two levels of abstraction lie in the degree of details expressed and in the depth of aggregation. For example conceptual architecture describes control and data and uses relationships between services categorized into domains, but concrete architecture defines strict interfaces and protocols used for communication between distributed and local component interfaces.

Mapping qualities to viewpoints is not straightforward. Simplicity is required in each view. Portability is covered mainly by the structural view, by a layered architectural style on the conceptual level and loosely coupled interfaces between layers and cohesive components on the concrete level. Integratability is mainly considered at the concrete level by most of the views. Maintainability has an affect on the structural and deployment views on both abstraction levels. Reuse of earlier developed components is considered in the development view. In addition each view includes design rationale that argues the decisions made during architecting.

3. Conceptual Service Architecture

While defining the conceptual structure of service architecture, the conceptual domain entities that are responsible for fulfilling the requirements are identified first. Second the responsibilities of these entities, computational and informational, are defined. Third the ways the identified entities use the properties provided by other entities are defined. A list or table is suitable for structuring information at this stage. Next graphical diagrams are used to ensure understandability, although the full benefit of graphical diagrams are not visible until the overall structures are stable and relations between entities are mature enough. Because the models on the conceptual architecture level are used for information sharing and promoting ideas of the architecture, it is essential that the models are easy to modify and maintain.

On the conceptual level stakeholders are primarily others than software engineers, therefore, simple models with intu-

FIGURE 3

Views on Two Levels of Abstraction in Service Architecture Modeling

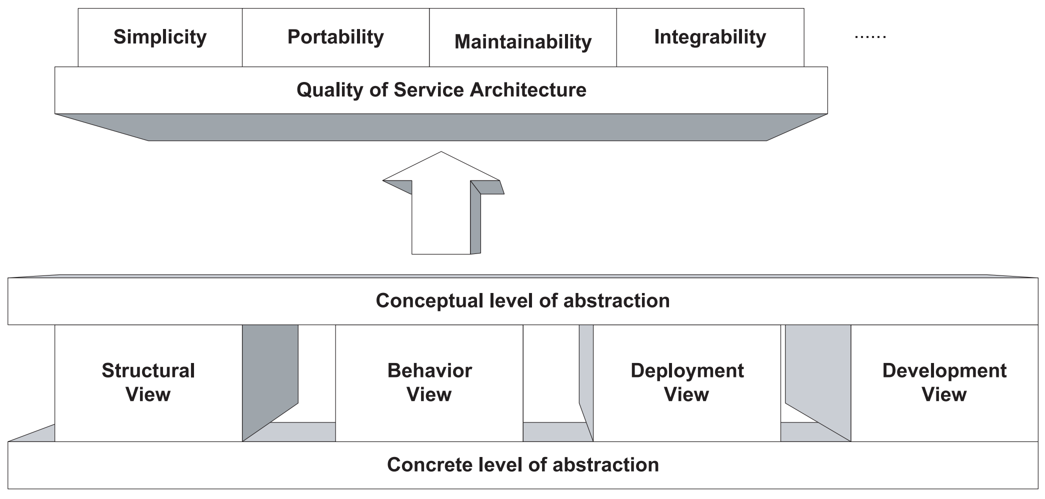


TABLE 2

Summary of the Elements of Conceptual Service Architecture

Name	Conceptual structure	Conceptual behavior
Concerns	What services and components are required, what are the responsibilities of services, how are quality requirements met?	What kinds of action does the service architecture provide for applications, which services collaborate in each action, how are the actions related to each other?
Stakeholders	System architects, service developers, product architects and developers, and maintainers	System architects, component designers, and service developers
Artifacts	Decomposition model including context diagram of the networked environment, computational structure and (domain) information structure; Table of clustered functional responsibilities, table of quality attributes	Table of interaction scenarios with services and UML collaboration diagram(s)

Name	Conceptual deployment	Conceptual development
Concerns	Which kinds of nodes are there in a system, what services have to be in the same unit of deployment, how can services be allocated to nodes?	What services and components does the company develop and what services are acquired from third parties, who is responsible for a service, which standards and enabling technologies do the services use?
Stakeholders	Service users and service developers	Project manager and component acquisition staff
Artifacts	UML deployment diagram	Business context model and explanations of the roles and relationships in a table.

itive notations are required. The conceptual architecture has to be understandable to various stakeholders. Managers and marketing staff in particular prefer a larger picture of the system and models that contain information relevant to them.

Structural views of software architecture are the most important. Design of the conceptual structure starts architecting by clustering the functional and quality requirements defined in the requirements definition phase and mapping them to the architectural entities, i.e., subsystems, services and components. Architectural styles are also selected in this phase.

3.1. The Conceptual Structural View

The conceptual structural view records the conceptual elements, the composition of the computational and informational entities inside each other, the interfaces between the elements, and the responsibilities the elements have in the system. In service architectures the purpose is to separate the computational structure and architectural relations from the structure of information shared by them (see *Table 4*).

Conceptual entities divide the software architecture into large functional and informational blocks. These blocks can be understood as systems, subsystems, services, or large-scale architectural components. Conceptual entities can also, however, be abstract categories of functionality, e.g., domains of services. Relationships define the interfaces between the entities. Composition is one relation, others are defined as types of relationships, namely data, uses and control relations that are used in the computational structure contrary to the information structure that uses the is-a and has relations. Types of relations are used in an attempt to avoid freezing design decisions too early, i.e., to be flexible for modification during alterations.

The computational structure is a decomposition model, using UML structure diagrams. Decomposition is natural as a result of the breaking down of the functional and information properties. Decomposition supports readability and modifiability, contrary to the hierarchical diagrams that lose their benefits if information required in modifying models is hidden. Therefore composition is presented in a single diagram if possible. Furthermore the use of inheritance shifts the focus away from defining the relationships and responsibilities of architectural entities, which is the main goal in conceptual architecting.

The UML package symbol is used to represent the high level conceptual entities. It is also, however, the symbol for low-

TABLE 3

Summary of the Elements of Concrete Service Architecture

Name	Concrete structure	Concrete behavior
Concerns	What are the concrete components needed for a corresponding conceptual component, what are the interfaces needed, how do services communicate with external actors?	How does a concrete component behave and respond to an event, what is the behavior of a set of concrete components?
Stakeholders	Component designers, service developers, and product developers	Component designers, testing engineers, and integrators
Artifacts	Hierarchical structural diagrams including information structure as class diagrams, intercomponent diagrams and intracomponent diagrams of concrete components; Component responsibilities as a list or a table	State diagrams and message sequence diagrams
Name	**Concrete deployment**	**Concrete development**
Concerns	What nodes and devices are there in a system and what do they have to do, what concrete components are allocated to each node and device?	What is the realization of a service or a component, how does a service or set of services relate to each other, how could a service be configured?
Stakeholders	Integrators and maintainers	Product developers and assets managers
Artifacts	Extended deployment diagram	Table of software component realizations and interface definitions (a list or a table)

TABLE 4

Structural Elements, Their Types, and Responsibilities on the Conceptual Leve

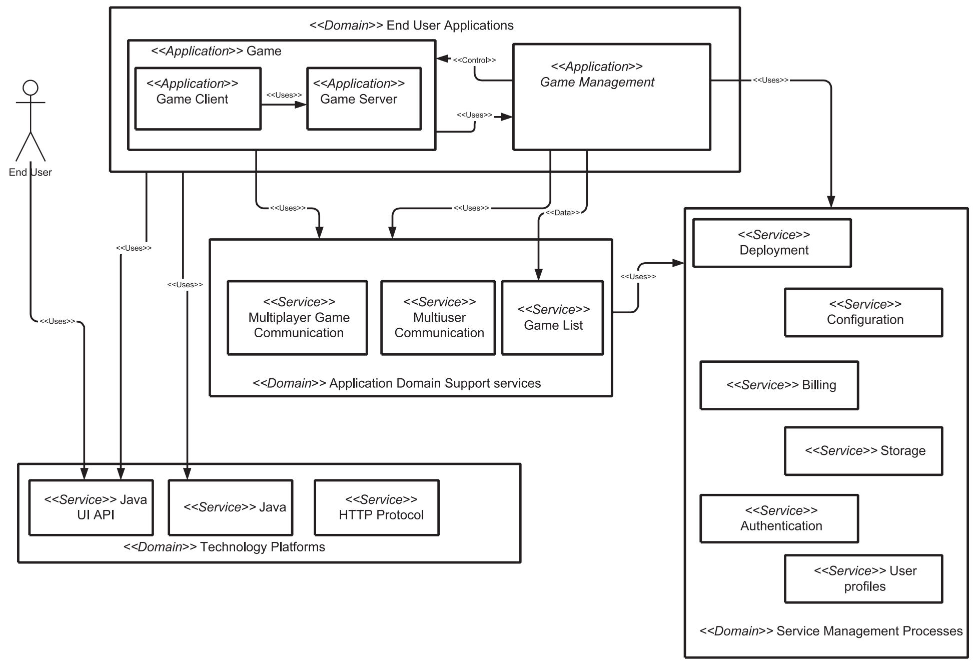
Element	Types of an element	Responsibilities of an element
Conceptual computational structure	Domain Service Component	What does the element have to do?
Conceptual information structure	Class Object	What information does the element require?
Conceptual context	System Node Network	What is the execution environment of the structural elements?
Conceptual structural relationship	Passes-data-to Passes-control-to Uses Is-a Has	How does the element relate to other elements?

level entities. In practice the use of the same symbols on two levels of abstraction, conceptual entities and design-level UML diagrams, confuses stakeholders, and that is why a slightly different visual look and feel on different abstraction levels increases readability and understandability of the architectural descriptions. *Figure 4* describes the conceptual computational structure of the game example used as a pilot in the WISE project.

The information structure that is shared between conceptual entities is described with class diagrams (see *Figure 5*). Here the benefits of object-oriented design and analysis are useful. The information is best modeled with class and object diagrams. Inheritance and is-a has and other similar relations between classes are appropriate and important. The types of relations, however, should be kept simple, and implementation-specific information should not be used on the conceptual level.

FIGURE 4

An Example of the Computational Structure of the Game Service



<<Domain>> End User Applications
<<Application>> Game
<<Application>> Game Client
<<Uses>>
<<Application>> Game Server
<<Control>>
<<Uses>>
<<Application>> Game Management
<<Uses>>
End User
<<Uses>>
<<Uses>>
<<Data>>
<<Uses>>
<<Uses>>
<<Uses>>
<<Service>> Deployment
<<Uses>>
<<Service>> Multiplayer Game Communication
<<Service>> Multiuser Communication
<<Service>> Game List
<<Service>> Configuration
<<Domain>> Application Domain Support services
<<Service>> Billing
<<Service>> Storage
<<Service>> Java UI API
<<Service>> Java
<<Service>> HTTP Protocol
<<Service>> Authentication
<<Domain>> Technology Platforms
<<Service>> User profiles
<<Domain>> Service Management Processes

3.2. The Conceptual Behavior View

The behavior viewpoint presents and justifies the selection of essential use cases and their clustering to the collaboration diagrams. The detailed interfaces are not yet defined in the conceptual structure and that is why behavior is modeled with collaboration diagrams. The messages should be abstract descriptions, as collaborations are. The use of message sequence charts (MSC) might be possible, but they are more appropriate on the concrete level when concrete structure with detailed information has already been defined.

FIGURE 5

Conceptual Information Structure of the Game Domain

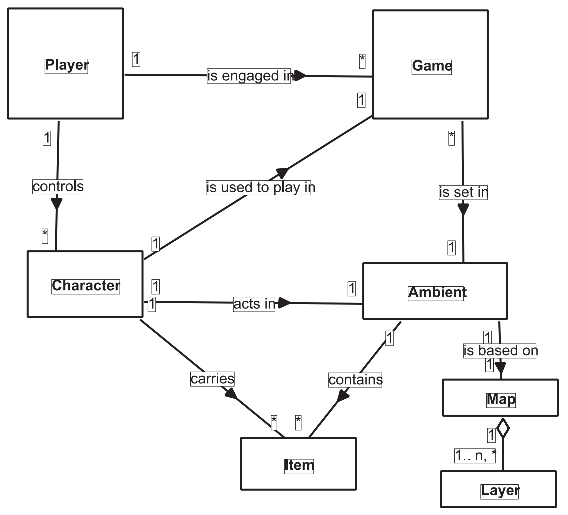


Figure 6 presents a collaboration diagram of the game service. The aim of the conceptual behavior view is to map only the essential use cases to the conceptual structure, and in this way, illustrate the behavior of the software service graphically. The main use cases may be clustered and presented in a single collaboration diagram.

3.3. Conceptual Deployment and Development Views

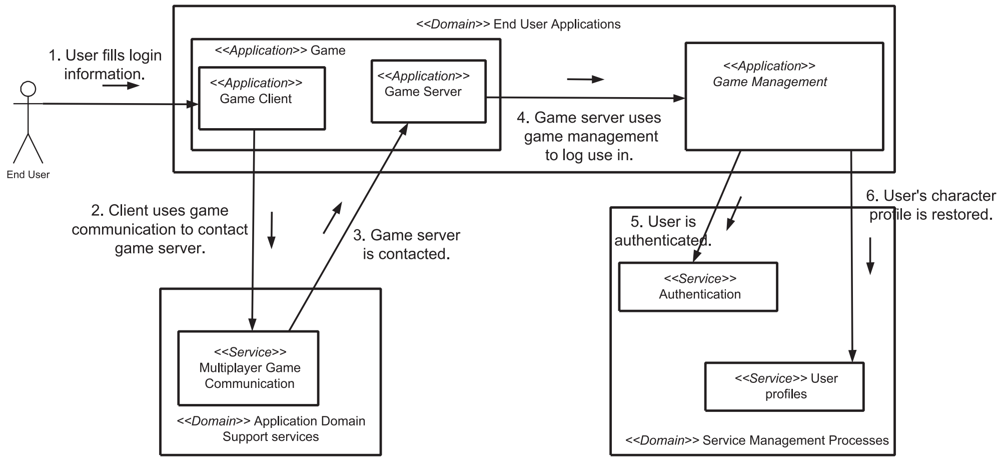
The deployment view that describes the allocation of conceptual entities with a UML deployment diagram is an essential part of service architecture. The conceptual deployment is defined for the processing nodes (see *Figure 7*). In order to maintain consistency, however, the same symbols as in the structure view should be used for the deployment entities. In *Figure 7* it is assumed that a game server node handles the management of a number of players with mobile devices and all synchronization and communication. The management services are most likely in a separate node

A similarly named conceptual element can be deployed both to the terminal and server side. This does not mean that the software elements or their responsibilities are the same. Such entities can be presented in separate deployment diagrams that make it easier to manage deployment diagrams.

The deployment viewpoint also includes an abstract business model. A business model describes "the various business actors, their roles, sources of revenues and links, interfaces and interaction between all the actors involved in the multifunctional environment" (Timmers, 1998). The busi-

FIGURE 6

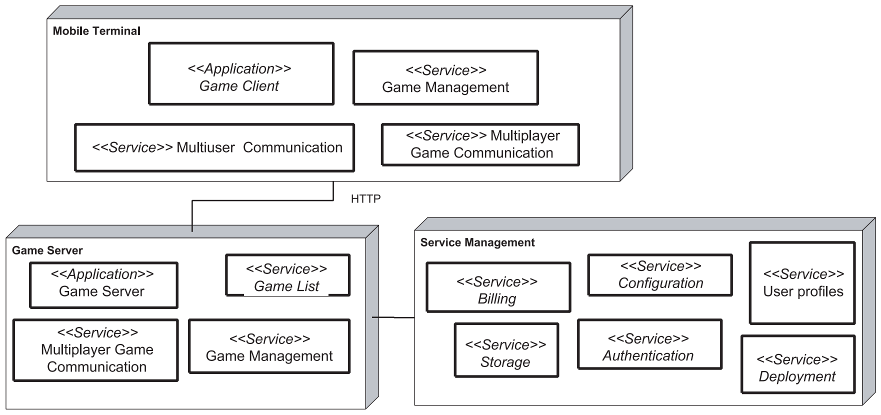
Conceptual Behavior as a Collaboration Diagram



<<Domain>> End User Applications
<<Application>> Game
1. User fills login information.
<<Application>> Game Client
<<Application>> Game Server
<<Application>> Game Management
4. Game server uses game management to log use in.
End User
2. Client uses game communication to contact game server.
3. Game server is contacted.
5. User is authenticated.
6. User's character profile is restored.
<<Service>> Authentication
<<Service>> Multiplayer Game Communication
<<Domain>> Application Domain Support services
<<Service>> User profiles
<<Domain>> Service Management Processes

FIGURE 7

Conceptual Deployment of the Game Service



Mobile Terminal
<<Application>> Game Client
<<Service>> Game Management
<<Service>> Multiuser Communication
<<Service>> Multiplayer Game Communication
HTTP
Game Server
<<Application>> Game Server
<<Service>> Game List
<<Service>> Multiplayer Game Communication
<<Service>> Game Management
Service Management
<<Service>> Billing
<<Service>> Configuration
<<Service>> User profiles
<<Service>> Storage
<<Service>> Authentication
<<Service>> Deployment

ness model represents the formalization of the business roles played by a group of stakeholders carrying out commercial relationships. The fundamental information technology (IT) aspect is the identification of which subsystems or components carry out which commercial relationships or which commercial relationships are delegated to a computerized system. Furthermore commercial relationships involve multiple and different business partners or different companies; hence they are subject to contractual agreements that formally identify all, which, and in which way the interactions among different companies must be accomplished.

The business model instantiated for the game service is depicted in *Figure 8,* in which only business roles and business relationships relevant to the service have been kept from the generic WISE business model. The shaded business roles play some task in the operation of the game service. The task can involve service provisioning (see the roles inside the dashed box) if there will be some software components deployed in a networked structure. The task does not involve service provisioning (see the roles outside the dashed box) if they have a business relationship prior to service provisioning, such as technology provider.

Relationship ApplicProv: This business relationship models the game download prior to game provisioning. Game download supports the acquisition from the user side of the application, i.e., client components, needed to play the game. Download can be carried out from both a fixed node, e.g. using any Internet browser, and a mobile node.

Relationship Peer: The game considers authentication and user profile storage as management services supported by a third-party service provider.

The development view should serve as a guide when considering the features required and provided by commercial off-the-shelf (COTS), original software component manufacturing (OCM), modified-off-the-shelf (MOTS), tailored, or new components. There is no clear choice in UML for this

FIGURE 8

Business Model of the Game Service

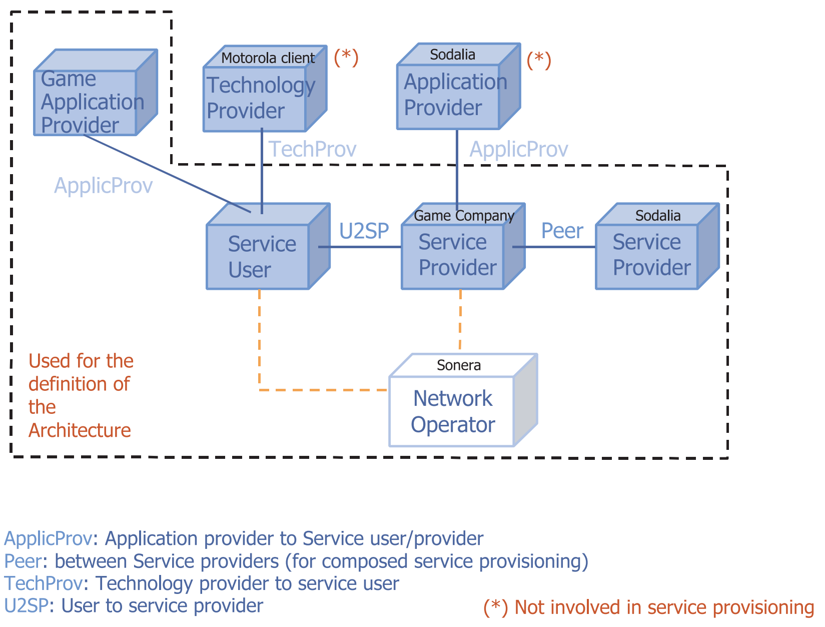
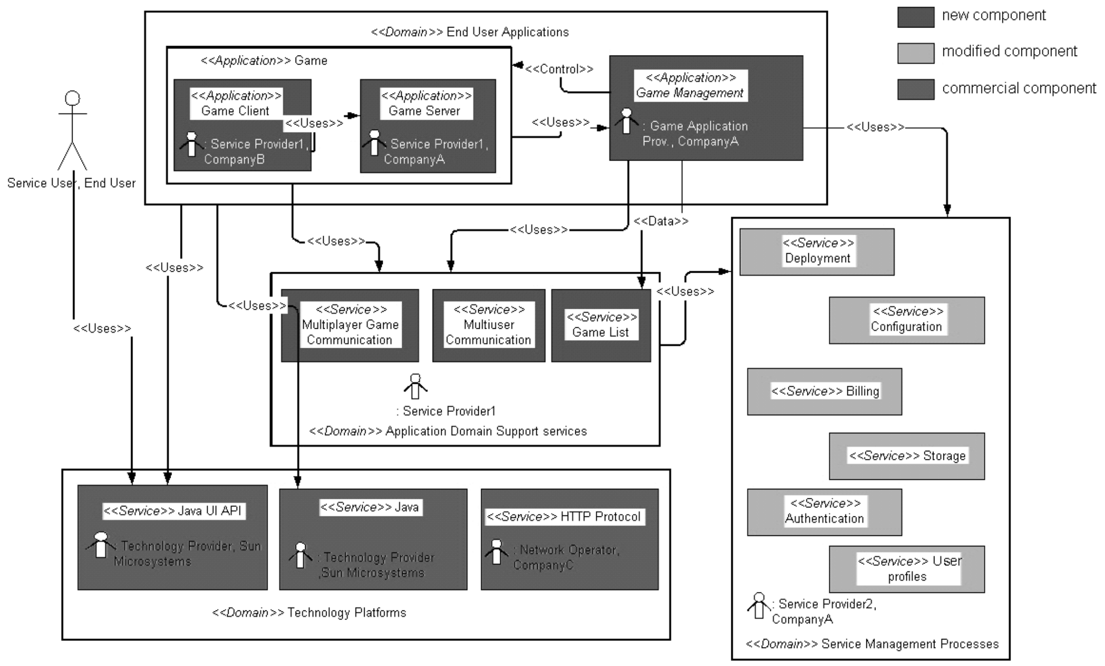

Game Application Provider
Motorola client
Technology Provider
(*)
Sodalia
Application Provider
(*)
TechProv
ApplicProv
ApplicProv
Service User
U2SP
Game Company
Service Provider
Peer
Sodalia
Service Provider
Sonera
Network Operator
Used for the definition of the Architecture
ApplicProv: Application provider to Service user/provider
Peer: between Service providers (for composed service provisioning)
TechProv: Technology provider to service user
U2SP: User to service provider
(*) Not involved in service provisioning

FIGURE 8

The Conceptual Development View Allocates Development Responsibilities


<<Domain>> End User Applications
<<Application>> Game
<<Application>> Game Client
: Service Provider1, CompanyB
<<Uses>>
<<Application>> Game Server
: Service Provider1, CompanyA
<<Control>>
<<Uses>>
<<Application>> Game Management
: Game Application Prov., CompanyA
<<Uses>>
new component
modified component
commercial component
Service User, End User
<<Uses>>
<<Uses>>
<<Uses>>
<<Data>>
<<Uses>>
<<Uses>>
<<Uses>>
<<Service>> Multiplayer Game Communication
<<Service>> Multiuser Communication
<<Service>> Game List
: Service Provider1
<<Domain>> Application Domain Support services
<<Service>> Deployment
<<Service>> Configuration
<<Service>> Billing
<<Service>> Storage
<<Service>> Authentication
<<Service>> User profiles
: Service Provider2, CompanyA
<<Domain>> Service Management Processes
<<Service>> Java UI API
: Technology Provider, Sun Microsystems
<<Service>> Java
: Technology Provider ,Sun Microsystems
<<Service>> HTTP Protocol
: Network Operator, CompanyC
<<Domain>> Technology Platforms

viewpoint. Color-coding for the different degrees of reuse can be used, as defined in *Figure 9*. Furthermore it should be easy to link the development view models to the structural view, so that the changes in structures are apparent in the development view and a separate description of structure is avoided.

4. Concrete Service Architecture

4.1. The Concrete Structural View

The concrete structural view defines the conceptual structure in more detail, including strictly defined component interfaces and patterns that are followed. The requirements for the notation of concrete architecture have already been fixed and initially applied in the context of various international service engineering projects (Lago, 2001 A and B). Further studies are still needed, however, especially when a quality-driven architecting approach is applied. *Figure 10* illustrates part of the structural view of the game service on the concrete level. The uppermost part of the figure describes the externally visible structure of components. These can export or implement multiple interfaces that can be distributed if remotely accessible by other components or local if private to the component itself. Distributed interfaces support distributed communication among components, whereas local interfaces define how the internal elements of a component are involved in local interactions.

The lower part of the figure describes the internal structure of a component, namely, how it is decomposed into component elements realizing the various interfaces. A component can be of two types: black box components represent those acquired from external sources, for example, commercial products or third-party components, and used or perceived by the system as opaque peers. On the other hand white box components are under development and therefore have a well-known internal structure.

The main contribution of the structural viewpoint is to provide an insight into software composition, which is especially important for distributed systems in which a component represents an atomic unit of distribution, and software distribution, i.e., the border between local and distributed subsystems. This requirement is achieved by making explicit (1) which components are parts of the system and which are external to the system, (2) which interfaces and supported interactions are distributed and which are local. The structural viewpoint also provides the building blocks on which the other viewpoints rely.

4.2. The Concrete Behavior View

The behavioral view defines how components interact to achieve the system's functionality. By identifying the dynamics of a system and the interactions among classes or among components, the behavioral viewpoint is based on sequence diagrams. Special attention will be paid to the specification of cross-components and intracomponents interactions. Cross-component interactions occur between different components and realize overall system functionality. Intracomponent interactions occur internally to a selected component and realize encapsulated implementation of a service offered to the external world. Diagrams providing the behavioral viewpoint belong to both the class and the instance spaces: class-level behavior is modeled using sequence diagrams, and whenever needed, instance-level sequence diagrams will show relevant example execution scenarios. The concrete behavioral view also defines rules for exceptions in communication protocols.

The behavioral view provides a two-step representation of how system functionality is realized in concrete terms: by separating intra- and intercomponent representation in both structure and behavior, a high-level system architecture is easily grasped from the intercomponent viewpoints, whereas a detailed system decomposition can be expanded in the intracomponent viewpoints.

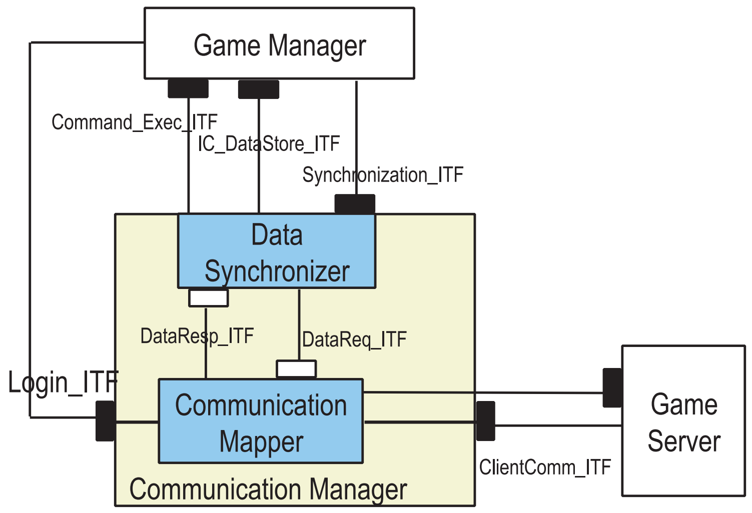
Also, by separating system behavior in the two perspectives, a compact model can be easily understood by non-technical stakeholders such as end users, thanks to a compact overall system representation, while component designers can use a detailed model.

4.3. The Concrete Deployment and Development Views

The UML Deployment Diagram maps the concrete components to the nodes of the execution environment. The diagram is also extended to map the deployment on the imple-

FIGURE 10

An Example of the Concrete Structural View of the Game Service



mented game specific business model, as depicted in *Figure 11.* In particular, the diagram evidences the following important issues:

- The domain associated with role of service user can be deployed on a fixed node, e.g., an Internet machine, for what concerns game download or on a mobile node, e.g., a cellular phone or any other mobile device.
- On the service provider side, there are two types of nodes mapped to two different domains playing the same role: the service node belonging to the provider of game control, and where the service core components are deployed, and the management service node belonging to the provider of outsourced management services, on which service-common components are located. In particular these components implement orthogonal services, such as user profile access and storage and authentication.

An additional advantage of the deployment viewpoint is that it provides a concrete analysis of security issues from architectural and business perspectives, and in addition to the usual implementation perspective, provides the necessary framework for adopting architectural standards. Furthermore the development viewpoint adds technology and implementation details. In particular it describes relevant aspects and constraints set by technologies deployed as black box components in the deployment viewpoint.

The concrete development view describes interfaces as abstract messages and parameters needed in component implementation. They are described in a separate document by tables in order to circulate documents according to work allocation between business stakeholders.

5. Quality Analysis of Service Architecture

Setting of the quality goals is essential in service engineering in order to reap the benefits of quality-oriented software development. Reasonable quality means that return on investment (ROI) should be considered for the different parts of software, i.e., the services used only in one application and the others used for a family of services. The reasoning rules of economic benefits might be input information from business models to the architecture development.

Although the pilot service is still under development, the initial analysis can be done based on the requirement specification and first draft descriptions of the service architecture. In the following sections the quality of service architecture is considered from the point of view of portability, maintainability, integratability, and simplicity.

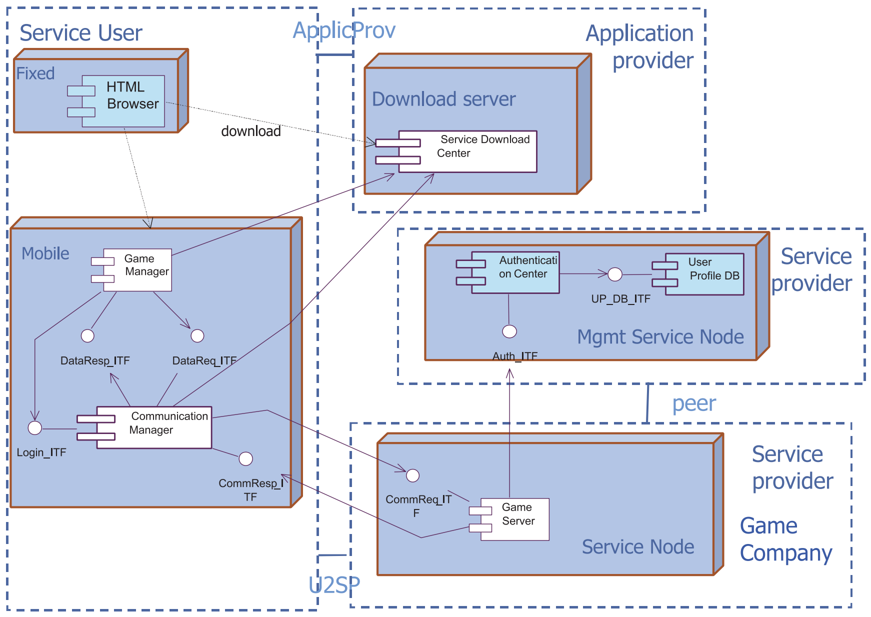
5.1. Portability of Services

Portability is the ability of the system to run under different computing systems: hardware, software, or a combination of the two (Dobrica, 2002). There are two issues that require portability: diversity of implementation technologies and diversity of communication technologies.

The architecture of the game service is heavily based on the existing Java technology; Java MIDlets on the client side and Java Enterprise Edition (J2EE) and Java Standard Edition (J2SE) on the server side. Although portability is not considered in the requirements specification, it can be seen from the architecture descriptions that game manager is the only component that is directly connected to the Kjava support component. Therefore portability of client software could be supported by a technology platform specific layer inside game manager that provides the required services for com-

FIGURE 11

Concrete Deployment View of the Game Service



munication, graphical user interfaces, and other elements. This layer has been defined on the conceptual level but is not visible on the concrete level. Thus the coarse-grained components do not give enough information on the concrete level of service architecture.

In the requirement specification of the game service, diversity of communication is defined as general packet radio service (GPRS) and UMTS. GPRS is used in the first phase, UMTS in the second or third phase. An initial communication description between the client and the server has been defined, but how the change over from GPRS to UMTS affects service development is not considered. Thus further exploration is needed in order to be able to anticipate architectural changes and isolate changing parts from the stable part of architecture. This separation makes it possible to identify generic components that can be utilized in several services and new features provided by UMTS can be quickly and easily utilized in wireless service engineering.

5.2. Maintainability of Service Platform

Maintainability is the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapted to a changed environment (Dobrica, 2002). As can be seen from the definition, maintainability is related to portability. It is a broader concept, however, considering the whole life cycle of a service and its execution environment.

The ease of making architectural modifications depends on at least the following prerequisites:

- Architecture is described in the same way; the meaning of terms and notation is shared and descriptions are made with the same accuracy.
- Architecture is carefully documented; the necessary information is available, dependencies between architectural elements have been identified and defined, traceability between descriptions is supported, and the reasons behind design decisions are documented.
- Implementation conforms to the architectural descriptions and defined standards.

Although maintainability is not mentioned as a quality requirement in the requirement specification and the architectural descriptions of the game service, we consider it through analysis of (1) the meaning of terms and notation and (2) dependencies and traceability. Conformance to the architecture can be analyzed once the implementation is ready.

In WISE we defined a short vocabulary and guidelines to how the pilot architectures should be defined. There are misunderstandings and weaknesses however. This may be the result of defective tutoring or unwillingness to change existing design practices or adapt the QADA method that has documented guidelines with examples, but it is a new method and approach in WISE. Because the terms, notation, and structure of the architectural descriptions form the common language with which architects communicate, more tutoring and communication of the method is required. The reason for the anomalies in notation was mainly the diversity of CASE tools utilized. A new tool, however, which was provided by the WISE architects, was applied, and notation guidelines were partially followed. In conclusion the map view of the architecture document was prepared in order to assist in following the guidelines. It seems that brief lists of instructions are more useful in practice than comprehensive guidelines, because they can be used and reread while architecting.

Concerning dependencies and traceability, two major weaknesses of the recent documentation are the incomplete interface descriptions of components and a lack of traceability between the viewpoints of the same abstraction level and between the conceptual and concrete levels. The aim is that the conceptual level captures commonalties and variables and provides overall information without technical details. This description is used for communication between heterogeneous stakeholders such as managers and developers. The objective of the concrete architecture is to provide component descriptions with strictly defined interfaces. The concrete architecture is used as a specification when the necessary components are allocated to the software developers inside or outside the organization, i.e., to be developed by the company itself, ordered from a subcontractor or bought from a commercial marketplace.

In summary, the following needs for improvements could be observed:

- Interfaces should be strictly defined in the concrete development view.
- Dependencies on selected technologies should be defined in a separate diagram in the development viewpoint.
- Mapping between abstraction levels is required. In order to address traceability to a greater degree, a separate view for mapping might be a better choice.
- A large number of message sequence chart (MSC) diagrams could be avoided by favoring collaboration diagrams and interface descriptions in a tabular form.
- The name of a component is part of its identification and therefore the name should be the same in every description reference.
- Design rationale is now part of each viewpoint but might be necessary in each diagram.

5.3. Integratability of Service Architecture

Integratability means the ability to make the separately developed components of the system work correctly together (Dobrica, 2002). Interoperability is a special case of integratability that measures the ability of a group of parts that constitutes a system to exchange information and use the one exchanged. Interoperability is omitted here because portability has a similar overall purpose. On the other hand integratability has been separated into two parts, namely horizontal integratability and vertical integratability. The purpose is to classify the service developers and their products into two categories: those that aim at global software markets with generic service products and those that provide customized services to end users.

In the game service horizontal integratability was considered on the conceptual level, but as the result of missing definitions of the services management service, only some observation can be made. The interface between communication manager and game server has been defined on the semantic level. In order to integrate the separately developed components, the interfaces and protocols need to be defined strictly. Vertical integratability was also difficult to see from the concrete architecture. An assumption was made that the communication manager provides a generic communication service to the game manager that is tightly coupled with the game application service. This means that the communication service might also be used in other services, but the game manager service must be developed separately for each game.

The following suggestions were proposed as improvements:

- Integration should be supported by a separate interface description with the protocol definitions in the development view.
- Integration interfaces should be generic and the first ones to be fixed in a service architecture.

The rationale for separated integration interfaces is that wireless service engineering is heavily based on the cooperation of several industrial partners that need flexibility in developing their own products but also strictly defined interfaces that are controlled by a coordination organization such as the Open Mobile Alliance (OMA[2]).

5.4. Simplicity of Architectural Descriptions

Simplicity can be defined as ease application and use of the architectural descriptions and therefore closely related to usability and reusability (Dobrica, 2002). Simplicity in this context, however, means the ability to use the platform services for different kinds of end-user services and add or create new platform services when richer applications require more powerful platform services or new adopted technology makes it possible to develop new support services or simplify their implementation.

In summary this ability was considered slightly on the conceptual level, but as the result of missing interface descriptions and application programming interface (API) for the game family this ability should be addressed much more in the next iteration phases. Therefore variability of services will be described in the most important services. The potential service categories that need variability support are the following: user interface services of mobile terminals, communication services, authentication services, and game application management services.

The results presented here are the first results from a pilot architecture that is not completely defined yet. It is obvious, however, that the application of the architecture-centric approach in wireless service engineering still requires further studies, improvements, and applications. That is why two other pilot architectures of different services will be constructed during the next two years. Furthermore all pilot architectures are developed incrementally which simulates the evolution of wireless services; the situation industrial partners encounter in wireless service business.

6. Conclusions

The aim of this paper is to justify the necessity of two separate levels of abstraction and the need for multiple viewpoints in architectural representations. These issues were argued with the different stakeholders and business roles related wireless service engineering. Second we emphasized quality as a key issue of wireless services and attested that quality has a different meaning depending on the role of the stakeholder. We defined the quality stack and applied it to illustrate the dependencies between qualities, stakeholders, and software components in the realization of a wireless service. As a link between end users' real added quality and the service architecture, we addressed the architecture-centric approach with the use of the QADA method that highlights how quality attributes are considered on different abstraction levels and how quality is carried through several development phases towards the realization of a wireless service.

Although the approach still needs further improvement, reassuring results have already been detected. During the first six months of the adoption of the approach to wireless service engineering, the common understanding of the meaning of service architecture has greatly increased and architectural descriptions have improved. Obstacles to the use of the WISE approach are partly organization-specific issues, such as earlier defined practices and tools and person-dependent issues such as available time to learn and take a new method into practice.

Acknowledgements

We would like to thank all partner members of the WISE project, especially Mr. Kalaoja and Mr. Tikkala who are the coauthors of the architecture guidelines and the game service architecture. Mr. Forchino and Mr. Tiella provided their insightful, practical knowledge about wireless service development as design descriptions of the game service and service management service for our use.

References

1. Bass, L., P. Clement, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley, Reading, MA (1998).
2 Bratthall, L., and P. Runeson. "A Taxonomy of Orthogonal Properties of Software Architectures," *Proceedings of NOSA'99*. University of Karlskrona (1998).
3. Dobrica, L., and E. Niemelä. "A Survey on Software Architecture Analysis Methods" *IEEE Transactions on Software Engineering*, vol. 28, no. 6 (July 2002): 638–653.
4. Hofmeister, C., R., Nord, and D. Soni. *Applied Software Architecture.* Addison–Wesley Longman Inc., Reading, 2000.
5. IEEE Computer Society. *IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems.* IEEE Std–1471–2000, 2000.
6. Jaaksi, A., J-M. Aalto, A. Aalto, and K. Vättö. "Tried and True Object Development," *Industry-Proven Approaches with UM,* 315. New York: Cambridge University Press, 1999.
7. Krutchen, P.B. "The 4+1 View Model of Architecture," *IEEE Software 12,* (1995): 42–50.
8. Lago, P., C.A. Licciardi, and A. Cuda. "Internet Boosts IN Towards New Advanced Services," *IEC Annual Review of Communications*, vol. 54, 2001.
9. Lago, P. "Rendering Distributed Systems in UML," *Unified Modeling Language: Systems Analysis, Design, and Development Issues*, edited by K. Siau and T. Halpin, Idea Group Publishing, 2001.

10. Matinlassi, M., E., Niemelä, and L. Dobrica. *Quality-Driven Architecture Design and Quality Analysis Method: A Revolutionary Initiation Approach to a Product Line Architecture.* VTT Publications 456, Espoo: Technical Research Centre of Finland, 2002.
11. Matinlassi, M., and J. Kalaoja. "Requirements for Service Architecture Modeling." To be published in *Workshop in Software Modeling Engineering of UML2002,* Dresden, Germany (Sep.30–Oct. 4, 2002).
12. Nenad, M. "Modeling Software Architectures in UML." *Workshop on Software Architectures and the Unified Modeling Language* (2000).
13. Purhonen, A., E. Niemelä, and M. Matinlassi. "Views of DSP Software and Service Architectures." Submitted to *Journal of Systems and Software*. 30.
14. Timmers, P. "Business Models for Electronic Markets." /netacademy/publications.nsf/all_pk/715__EM – Electronic Commerce in Europe_. EM – Electronic Markets, edited by Yves Gadient, Beat F. Schmid, and Dorian Selz, vol. 8, no. 2 (July 1998).
15. TINA Consortium Service Architecture specification. http://www.tinac.org.

Notes

1. This work has been partially supported by IST Project WISE (Wireless Internet Service Engineering), URL http://www.wwwise.org
2. http://www.openmobilealliance.org/overview.htm

PAPER VII

Platform services for wireless multimedia applications

Case studies

In: Ojala, T. & Ollila, M. (eds.). Proceedings of the 1st International Conference on Mobile and Ubiquitous Multimedia, MUM 2002. Oulu, Finland, 11–13 December 2002. Oulu: Oulu University Press. Pp. 76–81.

PLATFORM SERVICES FOR WIRELESS MULTIMEDIA APPLICATIONS: CASE STUDIES

Aki Tikkala and Mari Matinlassi

Software Architectures Group, VTT, Finland
{aki.tikkala, mari.matinlassi}@vtt.fi

Abstract. Today's multimedia is related to the use of computers to present multiple types of media in an integrated way. This paper introduces two case studies in service platform development for multimedia applications. The first one provides a streaming service, and the second one includes instant messaging and presence services for various types of multimedia applications. Although having different functional requirements, both the platforms conform to similar architectures because of convergent quality requirements: modifiability, integrability and portability. The dominant architectural styles used were blackboard and layered styles, reaching the qualities mentioned above. We proved that the selected styles achieved the selected quality requirements and moreover we realized quality attribute conflicts between functional and non-functional qualities, e.g. between performance and portability. Furthermore, we found out that design level choices can affect software quality. At least interoperability, simplicity and maintainability are influenced at the design level. Although service platforms are uncommon in the industry nowadays, it is expected that service platforms will increase their importance in multimedia application development in the near future.

1 Introduction

Multimedia means multiple types of media, e.g. text, graphics, video, animation, and sound, used in an integrated manner. To be more specific, multimedia is today, without exception, related to the use of computers to present media in an integrated way. The performance of electronic devices has improved, while the price of devices has decreased and therefore, multimedia and the transferring of multimedia are now commonplace.

Multimedia may be transferred through the network in two different ways, by downloading or by streaming. In *downloading*, the multimedia data is copied from a source to a destination as a whole, and not displayed before the download is complete, whereas with *streaming*, data display can be started before the entire file has been transmitted. Therefore, the choice is heavily affected by the technical facilities at hand. In a wireless environment, streaming is currently the typical choice as the devices have very limited capabilities for media storage.

Instant messaging is transferring messages between users in near real-time. The messages are usually, but not required to be, short media messages, preferably text. Instant messages are often used in a conversational mode, which means transferring messages back and forth fast enough for participants to maintain an interactive conversation. The *Presence* service is defined as a subscription to and notification of changes in the communication state of a user. The communication state consists of a set of communication means, address, and status for that user. A presence protocol is a protocol for providing such a service over the Internet or any IP network. Previous definitions of instant messaging and presence are results of the work in progress at the Internet Engineering Task Force (IETF).

By a multimedia service platform, we here mean software that is specially designed to provide generic lower level services for multimedia applications. By application, we mean e.g. games or chatting applications that again *utilize* generic services e.g. instant messaging and multimedia streaming. The successful multimedia service platform has well defined and described software architecture. As defined in [5], software architecture is the fundamental organization of software embodied in its components, their relationships to each other and to the computing environment. Software architecture also includes the principles guiding its design and evolution.

Software architecture closely influences software quality, i.e. the total of an entity's characteristics that contributes to its ability to satisfy stated and implied needs [4]. The main quality attributes of software architectures are described in [3] and among them are the driving quality requirements for the service platforms in the case study: *modifiability*, *integrability* and *portability*. Modifiability means the ability to make changes quickly and cost-effectively while integrability is the ability to make the separately developed components of the system work correctly together. Finally, portability is the system's ability to run under different computing systems: hardware, software or combination of the two.

This paper is structured as follows. First, we introduce two service platforms for multimedia applications. The first is a platform for applications utilizing streaming multimedia and the other provides instant messaging and presence services. Both the service platforms are based on Session Initiation Protocol (SIP) [8] or its extensions. SIP is an application-layer control protocol that can establish, modify, and terminate multimedia sessions (conferences) such as Internet telephony calls. Second, we compare the requirements and architectures of these two platforms and discuss the lessons learned during the platform evolution. In the end, we sum up the quality requirements and solutions gained in the case studies and draw up our future interests.

2 Case studies

2.1. Concepts and notation

Both the case studies discussed in this paper use a simple architectural notation. Squares represent *functional entities*, triangles are *interfaces* and lines between interfaces are *logical associations* among entities. Functional entities are differentiated with UML stereotypes. The cases introduced in this paper, use stereotypes as follows:

- **Domain.** *Domain* refers to the conceptual space of applications or subsystems that will be "covered" by a collection of reusable assets [6]. That is, domain is a common title for a group of software components. The domain component *itself* does not contain functionality, as a subsystem does.
- **Component.** *Software component* is a unit of composition with contractually specified interfaces and explicit context dependencies only [12].
- **Active object.** Active object is a special type of software component that is able to act without an external invocation.

An *interface* defines a contract between two components: a component requiring certain functionality and a component providing that functionality. An interface represents a first-class specification of the functionality that should be accessible through it [2]. The notation used here expresses interfaces as triangles. In the case of a required interface, the triangle points out of the component, whereas in a provided interface, it is vice versa.

2.2. Case 1: Service platform for streaming multimedia

The first case was developed on two iterations as follows. In the first iteration, the implemented functionality provided a means of streaming static multimedia (e.g. video) from a fixed media server to a wireless device. Herein, a commercial media server was used as a fixed server, whereas in the second iteration, we had a custom-made media server implementation in a wireless device (i.e. a mobile server). The second and final iteration enables real-time video calls between two wireless devices (Figure 1). The platform utilizes three protocols to control and transfer the video stream. The logical session between the parties is established and turned down by using SIP as a signalling protocol. In addition to the session control, the logical video stream also needs controlling. RTSP (Real-time Streaming Protocol) [10] is used as a stream control protocol for e.g. changing media parameters and start/stop streaming. Finally, using RTP (Real-time Protocol) [11] as a transfer protocol transfers the actual video data is transferred.

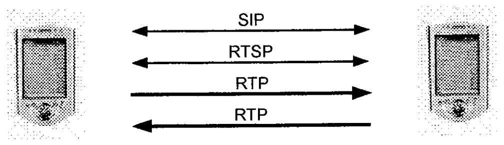


Figure 1. Video call between two wireless devices.

Both the functional and quality requirements affect architecture design. The functional requirements and constraints for the platform are listed below and Table 1 describes the quality requirements. The quality requirements in Table 1 are not the requirements for the whole platform. Instead, requirements are defined with more exact and system specific descriptions of the quality attribute in question [7]. The functional requirements of the service platform are:

- User is able to interact with the platform.
- Signalling protocol locates the receiver of the call and sets up the session with the receiver.
- Stream control protocol negotiates media parameters with the peer and establishes a media session with it.
- Platform supports data- reading from video camera.
- Video data is compressed to the MPEG-4 format.
- Media data is transferred from peer to peer.
- Video data is decoded before presentation.

Constraint:
- The implementation also has to be compact and efficient to be also usable in embedded devices.

Table 1. Quality requirements for case study 1.

Requirement	Definition
Modifiability	Easy and flexible adding (e.g. QoS protocols) and modification of the features should be considered.
Integrability	The platform is intended to be part of a wider multimedia platform rather than a stand-alone application. Thus major emphasis should be placed on the ease of integrating the platform into the existing products.
Portability	The platform should be easy to port to different operating environments.

The architecture of the streaming multimedia platform is based on two styles – the layered style and blackboard. The layered architecture style decomposes the system into a set of horizontal layers where each layer provides an additional level of abstraction over its lower layer and provides an interface for using the abstraction it represents to higher-level layer [2]. The layers are decomposed into a set of smaller components. The emphasis on this paper is on the streaming domain (Figure 2). The streaming subsystem is based on the blackboard architecture style. The blackboard architecture style contains an active data repository, which sends notifications to the clients that handle the data [1]. One of the benefits of the blackboard style is its adaptability to changing requirements. With the blackboard architectural style it is easy to add or remove a certain *type of data*. Furthermore, it is easy to add or remove the *number of instances* of this type. That is, processing components do not have explicit dependencies between each other, i.e. the control component captures the dependencies and is the only component that needs modifications if changes are incorporated in the processing component set [2].

In addition to the driving quality requirements defined in Table 1, interoperability and *reusability* also affected the platform design. The best way to achieve this is to follow protocol specification as closely as possible. In the domain of multimedia, the actions of standardization parties should be considered. Reusability is a nice-to-have quality and not defined in any more detail than as "the platform itself or parts of it should be reusable between different products or companies and/or between different releases of the same product".

In the architecture of the multimedia streaming platform an active object called Streamer corresponds to the active repository in the blackboard style. The Streamer invokes the independent event-based protocol components to handle the data coming from the network or from the camera. Protocol components do not have dependencies between each other's and they are stateless, i.e. components do not internally store any knowledge about their previous actions. Instead, the state of each protocol is stored to the Streamer component, which controls the sessions on behalf of the protocol components.

This kind of arrangement assigns a great deal of responsibility to the Streamer component. It has the task of controlling the whole streaming domain: handling the camera and the network, interacting with the user interface and controlling the set of protocol components. Although the responsibilities of the Streamer component are diverse, it provides relief for the other components. Protocol components have no other responsibilities than their core functionality required by their specification. All the bindings to the environment-dependent entities are in the Streamer, so all the compromises and the "glue-code" can be left to the Streamer.

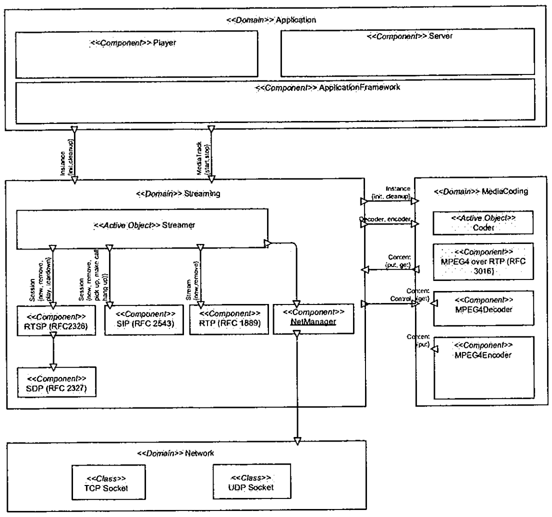


Figure 2. Architecture of the multimedia streaming platform.

2.3. Case 2: Service platform for instant messaging and presence

The second case study is a service platform that provides instant messaging and presence services for various kinds of multimedia applications. However, the instant messaging and presence services here cannot necessarily be considered as multimedia services, rather, they are services that *enable* the development of multimedia services. Functional requirements and constraints are listed below, and quality requirements are shown in Table 2.

The concept of a presence service is presented in IETF's draft 'Common Presence and Instant Messaging' (work in progress). The presence service consists of three entities – a presence user agent (PUA), a presence agent (PA) and a presence server. The Presence User Agent manipulates presence information for a presentee (i.e. a user). The presence Agent is responsible for receiving the subscription requests, responding to them and generating notifications of changes in the presence state. Usually, the PA agent is located in the Presence Server.

The platform is able to:

- notify the Presence Server about the terminal's/user's state,
- subscribe a buddy's presence information from the Presence Server,
- fetch a buddy's presence information from the Presence Server, and
- send one-to-one messages.

Constraint:

- When the project was started there was no SIMPLE (SIP for Instant Messaging and Presence Leveraging Extensions) [9] compatible server available. Because of this, the Presence Agent is included in the client.

The architecture of the instant message and presence service platform is presented in Figure 3. The architecture follows the blackboard-style and the layered style presented earlier. In this case, the IMManager component corresponds to the data repository and the SIP/SIMPLE component corresponds to the client in the blackboard style.

3 Discussion

The aim of case 1 was to study the domain of multimedia streaming with an experiment. While the first case was a pilot in the domain, the purpose of case 2 was more advanced; to apply the domain knowledge gained in industrial product development. The architecture of the first case was developed from scratch, and the knowledge gained in the first case was applied in the second one. Next, we describe the experiences and sum up the lessons learned during the case evolution.

The driving quality requirements for both cases were the same: modifiability, integrability and portability. Instead, the functional requirements were greatly *different*. That is, the functionality provided for the end user was different in case 1 and case 2, although the same protocol (SIP) was used.

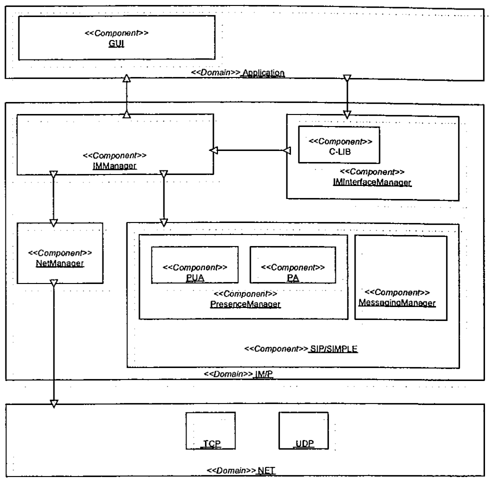


Figure 3. Architecture of the instant message and presence platform.

Despite the differing functional requirements of the cases, the architectural solutions were similar. This confirms the fact that quality requirements are the major driving force in (service platform) architectural design. In particular, the qualities of modifiability and integrability influenced the selection of blackboard style. Modifiability means e.g. switching responsibilities between a client (i.e. protocol component) and the data repository. It can be said that integrability follows modifiability, because through easy modifications it is also easy to integrate the whole platform to other vendors' software. If the driving quality attribute were performance, the selection of data-flow style, i.e. pipes-and-filters, would be a better choice.

Table 2. Quality requirements for case study 2.

Requirement	Definition
Modifiability	Architecture should enable the flexible integration of new instant messaging protocols. The interface between the service platform and GUI is not fixed to any implementation technique. Architecture should support both the two presence server location types: local and remote Presence Agents (PAs).
Integrability	The platform is intended to be usable in stand-alone application as well as a part of a wider multimedia platform.
Portability	Platform should be easy to port to different operating environments, graphical user interfaces or network implementations.

Table 1. Quality requirements for case study 1.

Requirement	Definition
Modifiability	Easy and flexible adding (e.g. QoS protocols) and modification of the features should be considered.
Integrability	The platform is intended to be part of a wider multimedia platform rather than a stand-alone application. Thus major emphasis should be placed on the ease of integrating the platform into the existing products.
Portability	The platform should be easy to port to different operating environments.

The architecture of the streaming multimedia platform is based on two styles – the layered style and blackboard. The layered architecture style decomposes the system into a set of horizontal layers where each layer provides an additional level of abstraction over its lower layer and provides an interface for using the abstraction it represents to higher-level layer [2]. The layers are decomposed into a set of smaller components. The emphasis on this paper is on the streaming domain (Figure 2). The streaming subsystem is based on the blackboard architecture style. The blackboard architecture style contains an active data repository, which sends notifications to the clients that handle the data [1]. One of the benefits of the blackboard style is its adaptability to changing requirements. With the blackboard architectural style it is easy to add or remove a certain *type of data*. Furthermore, it is easy to add or remove the *number of instances* of this type. That is, processing components do not have explicit dependencies between each other, i.e. the control component captures the dependencies and is the only component that needs modifications if changes are incorporated in the processing component set [2].

In addition to the driving quality requirements defined in Table 1, interoperability and *reusability* also affected the platform design. The best way to achieve this is to follow protocol specification as closely as possible. In the domain of multimedia, the actions of standardization parties should be considered. Reusability is a nice-to-have quality and not defined in any more detail than as "the platform itself or parts of it should be reusable between different products or companies and/or between different releases of the same product".

In the architecture of the multimedia streaming platform an active object called Streamer corresponds to the active repository in the blackboard style. The Streamer invokes the independent event-based protocol components to handle the data coming from the network or from the camera. Protocol components do not have dependencies between each other's and they are stateless, i.e. components do not internally store any knowledge about their previous actions. Instead, the state of each protocol is stored to the Streamer component, which controls the sessions on behalf of the protocol components.

This kind of arrangement assigns a great deal of responsibility to the Streamer component. It has the task of controlling the whole streaming domain: handling the camera and the network, interacting with the user interface and controlling the set of protocol components. Although the responsibilities of the Streamer component are diverse, it provides relief for the other components. Protocol components have no other responsibilities than their core functionality required by their specification. All the bindings to the environment-dependent entities are in the Streamer, so all the compromises and the "glue-code" can be left to the Streamer.

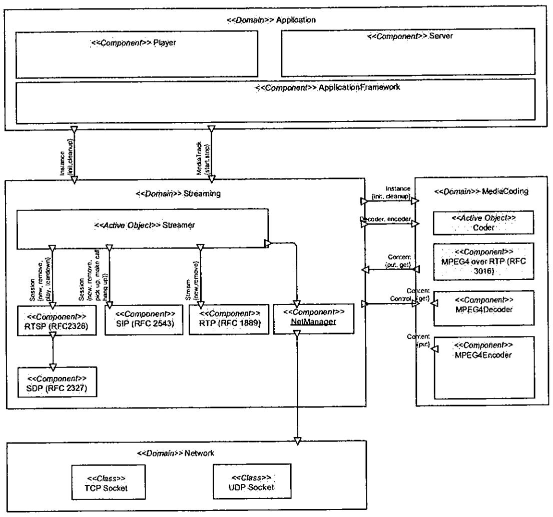


Figure 2. Architecture of the multimedia streaming platform.

2.3. Case 2: Service platform for instant messaging and presence

The second case study is a service platform that provides instant messaging and presence services for various kinds of multimedia applications. However, the instant messaging and presence services here cannot necessarily be considered as multimedia services, rather, they are services that *enable* the development of multimedia services. Functional

Second, the choice of layered style is obvious in service platform development. The layered style increases loose coupling through controlled inter-component interactions and easily achieves the qualities of modifiability and portability. For example, in case 2, porting from Windows environment to Linux took only an hour. This is because of two things. First, all environment- (e.g. operating system) dependent code was isolated to the two components – IMManager and NetManager. Secondly, all the dependencies to the upper or lower layers were created though these two components.

The lessons learned from case 1 were that blackboard style is a good solution in order to reach the qualities described above. Therefore, similar architectures were applied in both cases (i.e. reuse of domain knowledge). However, because of the proprietary software implementation, *source component* reuse was not allowed between the cases. Even though source component reuse was allowed, due to functional differences it would have been possible only for few percent of the code.

Secondly, the design of the first case satisfies us more than the second one. That is because the second case included a functional constraint that highly influenced the architecture: no server implementations were available. This causes the SIMPLE component not to be as pure as desired, i.e. the protocol component does not fully conform to the protocol specifications, because of the compromise caused by a functional constraint. This greatly hinders reusability.

Table 3 sums up the lessons learned along the platform evolution. It lists the quality requirements, summarizes the solutions and states whether the solutions are made at the architecture level or at design level. As seen, it is not possible to reach all the qualities at the same time. That is, some of the qualities are *conflicting*. Conflicting quality attributes are e.g. performance and modifiability or performance and portability. If the one gets more attention, the other one suffers.

In addition to conflict between quality attributes, the common problem of protocol architectures is how to achieve *interoperability*. It is an essential quality attribute, if the requirement is to make the software interoperable with the implementations of the other protocol vendors. On the other hand, interoperability often threatens to fall short of protocol specifications. Furthermore, interoperability requires a careful and costly testing phase.

In the future, it would be interesting to first extend the case 1 platform with the quality of service protocols. QoS protocols were already relevant by the time of the development of case 1, but were excluded in order to reach simplicity. QoS protocol extensions would verify whether the architecture really is extensible and also increase the quality of service, of course. Concerning the second case, the presence service and instant messaging services are still on the edge of development. The open issue is whether these services are to be included in the third generation protocol specifications. If they were, it would mean a great expansion in the applications of service platforms, such as introduced here, and a broader user community for services supporting multimedia.

Table 3. Summary of quality requirements and solutions gained in the case studies.

Quality requirement	**Solution**	**Level of choice**	**Comment**
Modifiability	Blackboard style Layered style	Architecture	Quality conflict with performance
Integrability	Componentisation and well defined interfaces	Architecture	Also design level choices affect, e.g. implementation language.
Portability	Layered style, i.e. operating environment specific functionality is isolated as a separate component (loose coupling)	Architecture	Quality conflict with performance
Performance	Pipes and filters style	Architecture	Quality conflict at least with modifiability
Simplicity Maintainability	Centralised control	Design	Means *few* control components but *several* passive components with clustered functionality.
Interoperability	Implementation conformance to protocol specifications Exceptions to specifications, set by the other implementations, are studied and taken into account	Design	This quality is highly dependent on the quality of the other software (to be interoperable with)

4 Conclusion

The purpose of this paper was to introduce two case studies of service platforms for different multimedia applications. The first case was a platform that provides streaming multimedia service for applications and the second case implements instant messaging and presence services.

In addition to introducing the case studies, we compared the requirements and architectures of these two platforms. The main commonalities and differences were that the quality requirements of the case studies were *similar*: modifiability, integrability and portability. This resulted in the use of similar architecture styles in both cases; however the functional requirements were *differing*. The architecture styles used were blackboard and layered styles.

In the end, we discussed the lessons learned during the platform evolution. The selected styles proved to support the quality attributes as expected. Since in the blackboard style, the processing components do not have dependencies on each other, and therefore, components can be added and removed without having to change other processing components, the blackboard achieves the qualities of modifiability and integrability. On the other hand, the layered style expects controlled inter-layer interactions and easily achieves the qualities of modifiability and portability. We also noticed that it is not possible to reach all the qualities at the same time. Conflicts often appear between functional and non-functional quality attributes, e.g. performance and modifiability.

In addition to architectural level solutions to achieve software quality, we confirmed a few important design level choices. Interoperability often causes one to compromise with protocol specifications, which again decreases reusability. In addition to the design phase, interoperability affects the testing phase. Testing has to be done carefully and is therefore an expensive phase in protocol software development. Furthermore, in order to achieve a simple and maintainable platform, one also has to use centralized control at the design level. This means fewer control components but several passive components.

The use of service architectures is quite unusual in industry for now, because of multimedia service platforms and their architectures still being on the edge of development. In the future, it is expected that multimedia services will increase their popularity among domain experts and therefore increase their importance in the development of multimedia applications.

5 Acknowledgements

Architectural figures used in this paper use an architecture description stencil by Jarmo Kalaoja at VTT. We would like to thank Hantro Products (http://www.hantro.com) and Creanor (http://www.creanor.com) for providing the case studies. The publication of this paper has been supported by Wise IST-2000-30028 (http://www.wwwise.org).

6 References

[1] Bass, L., Clement, P. & Kazman, R. (1998) Software Architecture in Practice. Addison-Wesley, Reading, MA, 452 p.

[2] Bosch, J. (2000) Design and Use of Software Architectures: Adopting and evolving a product-line approach. Addison-Wesley, Harlow, 354 p.

[3] Dobrica, L., Niemelä, E., (2000) A Strategy for Analysing Product Line Software Architectures. VTT Publications; 427, Espoo, VTT Electronics, 124 p.

[4] IEEE Computer Society, Information Technology – Software Quality characteristics and metrics – Part 1: Quality characteristics and sub-characteristics. ISO/IEC 9126-1. 1996.

[5] IEEE Std-1471-2000. (2000) IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems, IEEE Computer Society, 29p.

[6] Jacobson, I., Griss, M., Jonsson, P., "Software Reuse. Architecture, Process and Organization for Business Success", Addison Wesley, 1997.

[7] Matinlassi, M., Niemelä, E. 2002. Designing High Quality Architectures. ICSE2002, Workshop on Software Quality. 4p. Szyperski, C. Component Software. Beyond Object-Oriented Programming. New York: Addison Wesley Longman Ltd. 1997

[8] Rosenberg J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., Schooler, E. (2002) SIP: Session Initiation Protocol. IETF RFC 3261

[9] Rosenberg J., Willis D., Schulzrinne H., Huitema C., Aboba B., Gurle D., Oran D. (2002) Session Initiation Protocol (SIP) Extension for Presence. IETF work in progress

[10] Schulzrinne, H., Rao, A., Lanphier, R. (1998) Real Time Streaming Protocol (RTSP). IETF RFC 2326

[11] Schulzrinne,H., Casner, S., Frederick, R. (1996) RTP: A transport protocol for real-time applications. IETF RFC 1889

[12] Szyperski, C. (1997) Component Software. Beyond Object-Oriented Programming. Addison Wesley Longman Ltd., New York

Published by

Series title, number and report code of publication

VTT Publications 608
VTT–PUBS–608

Author(s)
Matinlassi, Mari

Title

Quality-driven Software Architecture Model Transformation Towards automation

Abstract
Model driven software development is about treating models as first class design entities and thereby raising the level of abstraction in software development. A model is a simplified image of a system and, further, model transformation means converting one model to another model of the same system. Transformation is a key to model driven development while automation of transformation is one of the essential goals of model driven architecture (MDA), an initiative to standardize model driven development. Model transformation aims at automating the transition from business models to implementation models. In addition to model refinement, model transformations are used for improving models by restructuring, completing and optimising them.

Quality-driven software architecture model transformation (QAMT) denotes changing an architectural model according to changing or varying quality properties, wherein a quality property is a non-functional interest of one or more system stakeholders. In this dissertation, I examine QAMT automation, i.e. reducing the need for human intervention in QAMT. Therefore, the research question in this dissertation is "how to make automation of QAMT possible". This dissertation provides an answer to the research question by presenting a model to support QAMT automation. The model is derived from the experience gained in four industrial cases and in one laboratory case study. The model is written with Unified Modelling Language 2.0 and includes *activities* to describe the process of transformation and collaborating *actors* that execute the activities.

The goals of the model are (1) to describe transformation as completely as possible, (2) to provide support toward automation, (3) to stay independent of implementation technologies, (4) to be mature and validated and (5) to conform to standards. Transformation is described by presenting a marked model, a mapping and a transformation record, and transformation activities. While the QAMT model does not support total automation of all the activities, it does reduce the need for human intervention. The QAMT model shows good performance in platform independence and it is validated in five different cases. Finally, the QAMT model promotes understandability by following, e.g., the terminology and specification structures defined in the most important standards in the area.

This research introduces an automation model for quality-driven software architecture model transformation. So far, the research effort on model driven architecture has been focusing on automating vertical transformations such as code generation. The work in this dissertation initiates the automation of horizontal model transformations and suggests future research topics to accumulate the knowledge on the subject and again to derive fresh topics to explore and new ideas to experiment with.

Keywords
software architecture, quality-driven software architecture development, model-driven development, model-driven architecture (MDA), model transformation

ISBN
951–38–6848–6 (soft back ed.)
951–38–6849–4 (URL: http://www.vtt.fi/publications/index.jsp)

Series title and ISSN VTT Publications 1235–0621 (soft back ed.) 1455–0849 (URL: http://www.vtt.fi/publications/index.jsp)	Project number

Date August 2006	Language English	Pages 101 p. + app. 95 p.	Price D

Name of project	Commissioned by
Contact VTT Technical Research Centre of Finland, Kaitoväylä 1, P.O. Box 1100 FI-90571 OULU, Finland Phone internat. +358 20 722 111 Fax +358 20 722 2320	Sold by VTT Technical Research Centre of Finland P.O.Box 1000 FI-02044 VTT, Finland Phone internat. +358 20 722 4404 Fax +358 20 722 4374

VTT PUBLICATIONS 608 Quality-driven software architecture model transformation. Towards automation Mari Matinlassi

Software intensive products have won popularity in everyday life today. An increasing need for faster, cheaper and even more versatile software intensive products sets a real challenge for the software industry. The software industry is constantly looking for ways to improve the cost-effectiveness of software development and the quality of software products.

The dissertation summary presents a model for quality-driven software architecture model transformation (QAMT). QAMT denotes changing an architectural model according to changing or varying quality properties, wherein a quality property is a non-functional interest of one or more system stakeholders. The aim of developing the QAMT model is to promote automation of transformation and thereby making changing software architecture easier. Reducing the need for human interaction in transforming an architectural model improves the cost-effectiveness and quality of software products.

Tätä julkaisua myy	Denna publikation säljs av	This publication is available from
VTT PL 1000 02044 VTT Puh. 020 722 4404 Faksi 020 722 4374	VTT PB 1000 02044 VTT Tel. 020 722 4404 Fax 020 722 4374	VTT P.O. Box 1000 FI-02044 VTT, Finland Phone internat. +358 20 722 4404 Fax +358 20 722 4374

ISBN 951-38-6848-6 (soft back ed.)
ISSN 1235-0621 (soft back ed.)

ISBN 951-38-6849-4 (URL: http://www.vtt.fi/inf/pdf/)
ISSN 1455-0849 (URL: http://www.vtt.fi/inf/pdf/)