Proc. Estonian Acad. Sci. Eng., 2005, **11**, 4, 296–312
https://doi.org/10.3176/eng.2005.4.04

A stylebase as a tool for modelling of quality-driven software architecture

Janne Merilinna and Eila Niemelä

Technical Research Centre of Finland, P.O. Box 1100, 90571 Oulu, Finland; {Janne.Merilinna, Eila.Niemela}@vtt.fi

Received 4 August 2005, in revised form 16 September 2005

Abstract. The main goal of model-driven engineering (MDE) is to improve software quality by using models as a means of producing high-quality systems with decreased development costs. Thus MDE makes it possible to represent software solutions by models and to evaluate and maintain models instead of the source code. However, software quality, i.e. quality of models, depends on how the quality characteristics of a system or a systems family have been taken into account in the architecture development. The main contribution of this paper is a stylebase, a tool including architectural styles and patterns, and design patterns, intended to be used while architecting. The stylebase is an integrated part of the architect's tooling environment, providing support for the design of the architecture and for quality evaluation of architectural models. The stylebase can be used as a part of a commercial tool and as an independent component of a distributed software development environment with heterogeneous modelling tools.

Key words: model-driven engineering, architectural pattern, design pattern, software quality.

1. INTRODUCTION

Although model-driven engineering, or model-based software development as it has traditionally been called, has been studied and applied to industrial systems since the 1980s, in model-driven architecture development there is still a huge gap between academic research and industrial applications [1], and also between Object Management Group (OMG) standards [2] and commercial tooling realizations [3]. Architecture-centric software development presumes the use of models instead of the source code as the primary assets of software. Software quality can also be evaluated from architectural models instead of running systems. When MDE is working correctly, the software company employing it may achieve

prime quality products with decreased development costs. This is based on the assumption that software architecture is properly designed and quality issues are taken into account in the architecture development.

Software architecture is represented by views, models and diagrams. An architectural view is a representation of the whole system from the perspective of a related set of concerns [4]. There are several approaches to the design of software architecture, each of them concentrating on different views of architecture [5–8]. However, there is no agreement on a common set of views or on ways to describe the architecture. This disagreement arises from the fact that the need for different architectural views is dependent on various factors, such as system size, domain, and stakeholders needing these views. Thus, each view is represented by a predefined set of models or diagrams, and the models of a view depend on the stakeholders and on their abilities and preferences for architectural representations.

Modelling helps in achieving the desired system quality, if the quality requirements are defined in models. Our contribution is a tooling environment with a stylebase, designed to assist in applying quality- and model-driven architecture development in software engineering.

The stylebase includes a set of architectural styles and patterns, and also design patterns used in designing and evaluating the architecture. The architecture can be architecture of a single system or that of a family of systems. Although family architecture needs a special kind of notation for handling the variation in structure, behaviour and allocation [9], the main design activities are the same: 1) defining the quality goals, 2) representing quality requirements in architectural models, and 3) evaluating how the quality requirements are met in models. The aim of the stylebase is to assist the architect in selecting the styles or patterns that best provide the desired quality goals.

The structure of the paper is as follows. Section 2 introduces the definitions and the development of the quality-driven architecture. Section 3 introduces how styles and patterns are represented in the stylebase. Section 4 presents how the stylebase has been implemented. Section 5 presents a case study which shows how the stylebase can be applied in architecture development. Final remarks close the paper.

2. QUALITY-DRIVEN SOFTWARE ARCHITECTURE DEVELOPMENT

2.1. Definitions

Software architecture denotes a structure or structures of a system, which represent software components and their externally visible properties and relationships between them [10]. An explicit representation of software architecture has three purposes [11]: stakeholder communication, software product lines and quality attribute assessment.

Software architecture allows early communication between the stakeholders, involved in the software development process. The reviewing and accepting of architecture descriptions by stakeholders ensures that software development can proceed smoothly.

In a software product line, the architecture defines a common architecture and a set of components to be used in the entire product line. Choosing an appropriate architecture for a product line is essential because the quality of the family architecture reflects on all the product members included in the product line.

Software architecture embodies non-functional characteristics, which are also called quality attributes. There are two main categories of quality attributes [12]: execution qualities (e.g. performance, availability and reliability), which are discernible only at run-time, and evolution qualities (e.g. integrability, modifiability and maintainability), which are considered in architecture development and evolution.

Several attempts have been made for defining exact meaning of different quality attributes. The following definitions are based on the quality model and related literature [13,14]. *Availability* measures the proportion of time the system is up and running. *Reliability* is the capability of a software system or component to keep operating over time or to perform its required functions under the stated conditions for a specific period of time. *Integrability* is the capability to make separately developed components of a system to work together. *Extensibility* is the capability to allow new components and features to be added or existing ones to be updated by recompilation, reinstallation and dynamic configuration. *Modifiability* is the capability to make changes quickly and cost-effectively. *Maintainability* is the ease with which a software system or component can be modified or adapted to a changed environment. *Reusability* is the capability to reuse the system structure or some of its components again in future applications.

Quality attributes are realized in different ways in different kinds of architecture. Therefore, the styles and patterns used in architecture may promote realizing some attributes and prevent some others.

An architectural style is a description of component types and their topology, and defines the constraints on a set of architectures that satisfy them. Architectural style is not architecture, but it still conveys an image of the system [10].

When an architectural style is strictly defined, it becomes an architectural pattern. An architectural pattern expresses a fundamental structural schema for software systems, which are applied for high-level system subdivisions, distribution, interaction and adaptation [15].

A design pattern describes a recurring structure of communicating components, which solves a general problem in a particular context [16]. Thus design patterns are microarchitectures applied in particular contexts.

Idioms are programming language specific patterns defining how a particular architectural model has to be implemented in a specific language. Thus, they are patterns of the lowest abstraction level.

2.2. Quality-driven architecture design and quality analysis

The basic principle of modelling quality-driven architecture is to emphasize the importance of quality attributes during the architecture development. This means designing software architecture with specific patterns that promote specific quality attributes. The QADA® [17] methodology is an approach to designing software architectures from the quality point of view.

In QADA®, architecture is modelled from four viewpoints: structure, behaviour, deployment and development [8,17,18]. Architecture is also described on two levels of abstraction, conceptual and concrete, with similarly named views. These viewpoints embody the quality characteristics of the architecture. Qualities are only visible at the architectural level through these views and the models and notations used in them, as well as through the design rationale of each view.

3. STYLEBASE AS A PATTERN REPOSITORY

In QADA®, the knowledge of patterns has been gathered into a single knowledge base, called stylebase. The intention is to move from the notion of architectural styles to the ability to deduce the patterns that best meet the desired quality goals. The purpose of the stylebase is to make the architecture design process more systematic and more predictable.

3.1. Contents of the stylebase

The stylebase is a collection of informal aspects of patterns represented in a way that makes it easy to reason what pattern is the most suitable for the particular situation. Describing patterns informally as in [15,16] is insufficient. Patterns have to be defined as explicitly as possible. At the moment, the purpose is not to describe patterns in a formal way as in [19] but to depict the informal aspects of patterns in an explicit form. Thus, the stylebase contains the following information for each pattern: *name* of the pattern, *reference*, *abstraction level*, *diagram*, *purpose*, *quality attribute*, *component type*, *component role*, *connector type*, *data topology*, *control topology*, *guide* and *figure*.

The first parameter presents the *name* of the pattern. There may exist several patterns with the same name and a pattern may have more than one name. Therefore the second parameter, the *reference* of the pattern, is also stored into the repository to specify the pattern more accurately.

Abstraction level defines the level of abstraction on which the pattern occurs: patterns are thus identified as architectural or design patterns. *Diagram* describes the UML 2.0 diagram [20], e.g. class, deployment, or composite structure, in which the pattern is intended to be used. *Purpose* describes the primary problem for which the pattern is intended [15,16]. Thus patterns are categorized by the *purpose*.

The sixth parameter describes *quality attributes*, such as modifiability, extensibility and reliability, which the pattern promotes. A pattern can promote a set of quality attributes as presented in [12]. Furthermore, a pattern may also promote "anti-qualities", i.e. the pattern sets constraints on achieving some other qualities than what it supports. Therefore, anti-qualities are also stored into the stylebase.

As architectural structures are composed of components, *component type* [21] is also included in the stylebase. *Component type* describes which kinds of components there are in a pattern. At the moment, five component types have been identified: *data components* are often passive data storages, such as files and data bases. *Control components* master other components by invoking them or by controlling access rights. *Computation components* do the data processing work. *Interface components* provide an interface for other components. Interface components do not process any data themselves, nor do *package components*, which gather and categorize smaller components.

Because components are connected by connectors, the *connector type* is also included in the stylebase. *Connector types* describe in detail the communication between various components, involving messages, data streams, events and the like.

A pattern may show one type of the component with different kinds of behaviour. Thus, *component role* is included in the stylebase. *Component role* determines the role, e.g. server, client, blackboard, source, etc., that the component plays in the pattern, and thus the role determines the behaviour of the component to a certain degree.

The *data topology* and *control topology* parameters present what kinds of geometric forms data and control take in the pattern [21]. Thus, *data topology* and *control topology* also define the component topology. Shaw and Clements define five topologies: *hierarchical*, *linear*, *star*, *arbitrary* and *fixed*.

Guide is the last textual parameter in the stylebase. The *guide* serves the architect in a traditional way by providing an informal description of the pattern. In the case of architectural patterns, the *guide* contains the context, definition, structure and implementation of a pattern. While considering design patterns the *guide* also contains information about intent, applicability, participants, collaborations and consequences.

In addition to the textual properties, each pattern is illustrated through a *figure*. The *figure* presents how the pattern should look when implemented. Figure 1 depicts a conceptual data model of architectural patterns with cardinalities.

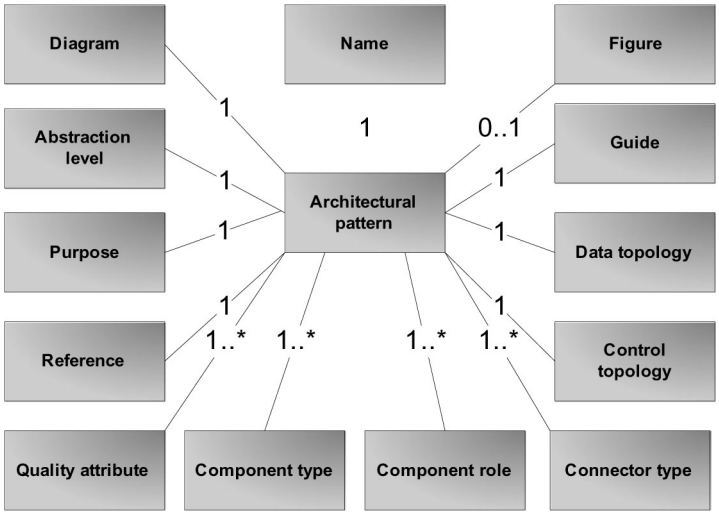

Fig. 1. Architectural pattern description.

3.2. Describing patterns in the stylebase

As an example of architectural patterns, Blackboard is introduced. According to Buschmann [15], the Blackboard architectural pattern "is useful for problems for which no deterministic solution strategies are known" and in which "several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution".

The idea of Blackboard is to have a collection of independent components, working cooperatively on a common data structure. Blackboard is a central data store, to which all the knowledge sources have access. The knowledge sources are independent subsystems or components that solve some specific aspects of the overall problem. The component, which organizes the whole system, is called control. The control component evaluates the current state of processing and coordinates the knowledge sources. Next, we illustrate how the patterns are described in the stylebase.

The *name* of the pattern is set as "Blackboard" and for *reference* we set "[15]" to make sure which "Blackboard" we are defining. As the Blackboard is an architectural pattern, the *abstraction level* is set to "conceptual". UML 2.0 provides a new diagram, i.e. the composite structure diagram, for describing architectural aspects. Thus, *diagram* is set to "composite structure". Buschmann et al. [15] categorize the Blackboard architectural pattern into the group "from mud to structure", thus this is also set for the *purpose*.

The set of quality attributes promoted by the Blackboard stems from the fact that the control component provides loose coupling and centralized control of the other components. The Blackboard architecture pattern promotes four *quality attributes*: availability, maintainability, modifiability and reusability [18]. Maintain-

ability, however, can be defined as a composite of other quality attributes [12] such as flexibility, reusability, modifiability, testability and integrability. Thus, maintainability can be considered as a category of quality attributes. In addition, maintainability is rather a system level attribute than an architecture level one. Therefore, maintainability is left out. However, as the control component provides loose coupling between components, adding new components to a system may require only minor modifications to the control and data component; it is considered that the Blackboard also promotes extensibility. No anti-qualities are set. Therefore, the final list of *quality attributes* takes the following form: availability, extensibility, modifiability, reliability and reusability.

The Blackboard pattern is well defined and formed. There is a list of components that must exist in the pattern. Three types of components are identified: control, which controls the other components, data component functioning as a blackboard, and knowledge sources doing the computation work. Therefore, the *component type* is an array of component types: “control”, “data” and “computation”. Similarly, “control”, “Blackboard” and “source” are set for the *component role*.

The communication between components is defined by communication participants. Control component controls the other components when the Blackboard is accessed with data signals. Thus, “control” and “data” are defined as a *connector type*.

The Blackboard is constructed by dividing the system to the defined specific components; the control is at the centre, the Blackboard at the bottom and the knowledge sources surround the control. The *control topology* clearly takes the form of a star. As the Blackboard is only accessed from the above and data is propagated only from top to bottom and back, the *data topology* takes a hierarchical form. Therefore, “star” is set for *control topology* and “hierarchical” to *data topology*.

Although the *guide* can contain miscellaneous text, a description, similar to the description of pattern catalogues, provides the most advantage for the pattern guide. The last parameter, *figure*, is also drawn and stored into the stylebase.

4. IMPLEMENTATION OF THE Q-STYLEBASE

The purpose is to enable an easy and fast way to browse patterns in the stylebase. This would not be possible without tool support. In this section, we take a closer look at the requirements of the stylebase and at the Q-Stylebase browsing tool, to be followed by a discussion of their design and implementation.

4.1. Requirements for the Q-Stylebase

The goal is to collect the design and architectural patterns into a single place, where patterns are easily accessed and always ready at hand for the architect.

Therefore, 1) patterns shall be easily accessed from different modelling tools, 2) the stylebase shall be easy to keep up-to-date, 3) it shall be possible for architects to share their own patterns with the others through the stylebase, and 4) architects shall always have up-to-date knowledge of available patterns.

Considering these requirements, the stylebase is designed as a database, which is accessible to the developers from commercial modelling tools through a local area network or from the Internet.

Keeping the stylebase always up-to-date is necessary as the software architects may find new ways of solving a common problem in their problem area and they may wish to bring this to the attention of the rest of the developers. Instead of updating the stylebase in every computer one by one, developers can share their newly discovered patterns with the others just by pressing a button as they all work on a common pattern repository. This allows the stylebase always to be kept up-to-date and every developer in the team to have the most up-to-date information about the available patterns.

The stylebase is used through a tool, called Q-Stylebase, for which three main requirements are set. The first requirement comes from the fact that in the near future the Q-Stylebase has been planned to be extended with new features. Thus, the tool should be extendable. In addition, the Q-Stylebase should allow integration with various modelling tools. These requirements have to be considered in the architecture development. The third requirement concerns the user interface of the stylebase. The stylebase should allow browsing by pattern name, quality attributes and abstraction level; i.e. the stylebase should allow choosing whether to browse design or architectural pattern, for example.

4.2. Design and implementation

The architecture of the Q-Stylebase is designed to provide loose coupling between components by hiding their implementation from each other. This is achieved by applying a well-known behavioural pattern called Mediator [16]. The Mediator pattern states that only two components know each other and all components interact among themselves through one central component. By applying the Mediator, the changed implementation of one component is not shown in other components. The loose coupling between components also facilitates implementing possible improvements and extensions.

Three task-specific components were defined: 1) graphical user interface, 2) mediator and 3) database handler. Visual C++ was chosen for implementing the user interface, as it was also used for other components.

The user interface of the Q-Stylebase was divided into two separate dialogues. The Query dialogue (Fig. 2) allows the architect to browse the stylebase by pattern name, quality attributes and abstraction level. Management dialogue (Fig. 3) shows more details of the pattern and enables performing basic database management functions.

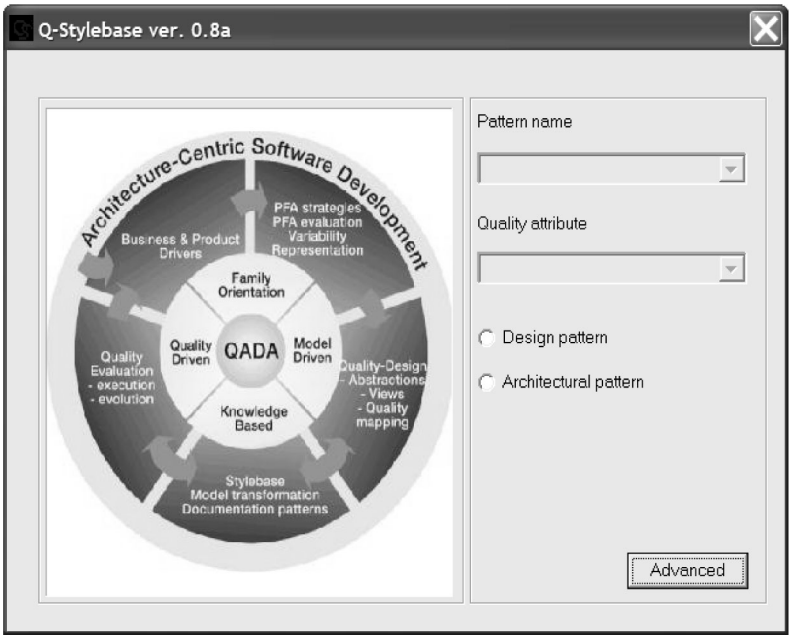

Fig. 2. Query dialogue.

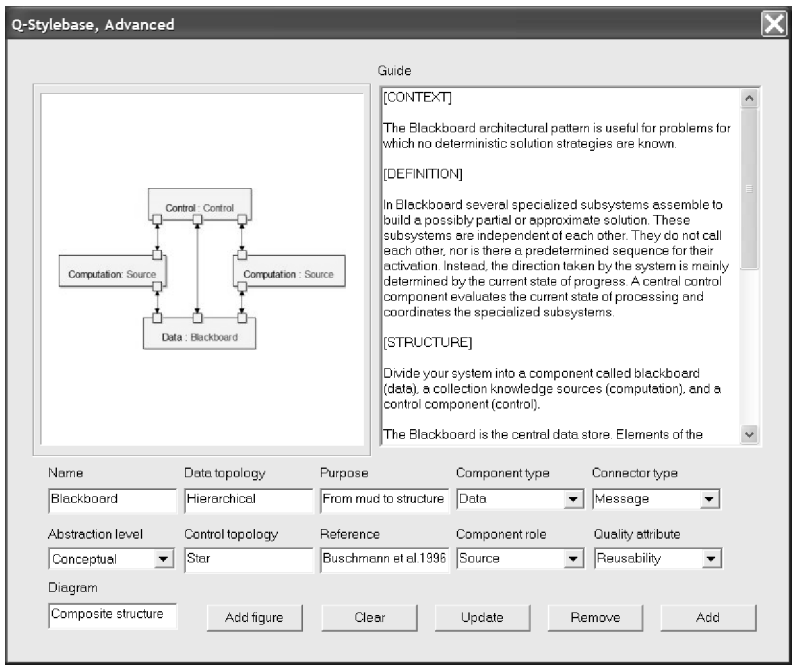

Fig. 3. Management dialogue.

One purpose of the Q-Stylebase is to provide an electronic library for patterns. Since modelling takes place in CASE tools, the Q-Stylebase was integrated to a commercial modelling tool. Telelogic Tau/Developer, which provides, among other things, a COM interface for third-party plug-ins, was selected to be used as the modelling tool [3]. In addition to a plug-in version of the Q-Stylebase, a stand-alone version was also developed.

As stated above, the stylebase was implemented with a database. For the database, MySQL [22] was chosen, as it was considered sufficient for our purposes and, above all, for the reason that it was distributed under the General Public License [23]. MySQL is a relational database using Structured Query Language (SQL) as the query language.

There are usually two means for accessing SQL databases from applications: 1) using a database-specific Application Programming Interface (API) or 2) accessing the database through an intermediate API such as Open Database Connectivity (ODBC). The latter is database independent, which means that using ODBC does not restrict the database used, and thus there is no need to change the source code if the database is changed. Thus, we chose the ODBC. The database handler provides basic database manipulation services such as querying, adding, removing and updating.

Figure 4 depicts the stylebase structure by an entity relationship diagram. The stylebase schema consists of five tables. The pattern table has ten attributes, with index as the key attribute. The component type, component role, connector type

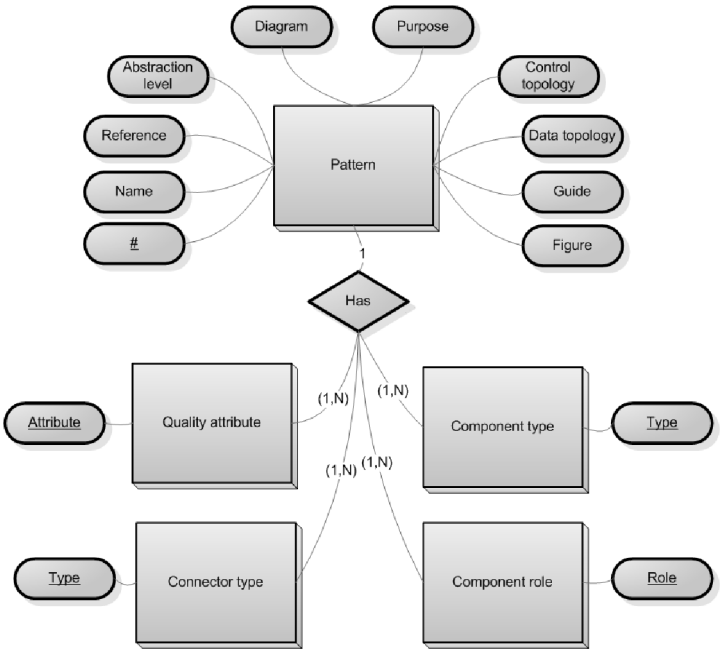

Fig. 4. Stylebase schema.

and quality attributes are set to separate tables. This arrangement permits a representation of several different attributes, e.g. several component types, for one pattern. It has to be noted that the tables also have their own key attribute but also a foreign to the pattern table. Thus, one pattern may not have several attributes of a single type, i.e. one pattern may only have one "data component" defined in the component type table.

As it is assumed that the stylebase will not grow extensively, rigorous database optimization was not considered necessary. The present stylebase version was designed from the viewpoint of usability.

As a result of the implementation, Q-Stylebase is available in stand-alone and plug-in versions, the main difference between these being the starting sequence. The stand-alone version is an executable. The plug-in version is integrated into the menu structure of Telelogic Tau/Developer from which it is thus also started.

5. APPLYING THE Q-STYLEBASE

Q-Stylebase can be used in three ways: 1) as an electronic library for patterns, 2) as a quality-driven architecture model construction guide and 3) as a quality-driven architecture model evaluation guide. When used as an electronic library, an architect browses the stylebase like ordinary pattern catalogues such as [15,16]. Next, the use of the quality assets is described in detail.

The Q-Stylebase provides means for outlining architecture candidates for the system to be developed. This is done by following the four steps (Fig. 2):

- Choose the pattern abstraction level in order to browse patterns at a specific level of abstraction.
- Select the desired quality attribute.
- Browse the pattern name list in order to reveal the patterns that promote the desired quality attribute. Choose one.
- The quality attribute list is updated with all the quality attributes that the pattern candidate promotes and constrains. Validate the other quality attributes.

Now, if the architect is not familiar with the pattern candidate, more details and guidance can be found by pressing the "Advanced" button. A new dialogue is shown (Fig. 3), containing the information necessary for constructing and using the pattern. The information presented assists the architect in applying the pattern.

In a situation where the architecture is already constructed, the evaluation of its quality assets can be performed with the help of the stylebase. Knowing the patterns used in the architecture model, the architect can browse the stylebase by pattern name and thus discover what qualities they promote and constrain. In this way, the architect can evaluate the quality properties of the entire architecture.

In order to illustrate how Q-Stylebase can be used for quality-driven software architecture modelling, a simple case study is presented. In this study, we show how to apply the Q-Stylebase for evaluating the qualities of a given architecture and how to redesign the architecture to match changing quality requirements.

5.1. Description of the case

The case is called Distribution Service Platform (DiSeP). The goal of the DiSeP case is to make the software components in a networked environment to interact spontaneously. The components in the DiSeP are various kinds of services which are either a part of the platform or a part of the application utilizing the platform. The configuration of the network may change dynamically. In other words, the number of modules or the range of the available services may vary. The main goal of the DiSeP is to maintain the interoperability of the services despite the dynamic nature of the network [17]. Figure 5 presents the conceptual architecture of the DiSeP system.

At the top of Fig. 5, the "Application" represents the application using DiSeP. The "Interface" layer below Application contains four interface components for services that can be directly accessed by the application. The layer below the interface layer contains two components: the "Lease service" for lease management and the "Directory service" providing a directory for distributed data storage. The most complex layer contains five components, which are responsible for receiving and processing incoming control information and sending outgoing control information. The "Activator service" monitors the state of the network, "Data storage" works as distributed data storage, "Interpreter" encodes and decodes XML messages, "Data distribution" operates the data storage and the "Location service" manages location information specific aspects. The last layer, the "Communication service" provides services that handle communication between different units in the network. It can be concluded that DiSeP applies the Layers architectural pattern in the most part.

5.2. Using the Q-Stylebase for refining the architecture

In this example, we present a simplified workflow for evaluating the quality properties of the architecture and redesigning the DiSeP architecture model to promote extensibility. In the case of DiSeP, the extensibility requirement is regarded as the capability of a system to acquire new service components without influencing other components at the affected layers. In brief, adding new components should not affect the Lease service, Directory service, Data distribution, Location service, Data storage, Activator service or the Interpreter. In Fig. 5, it can be discerned that this is not possible as the control topology is arbitrary. The impact of adding new services depends on the type of the added component; thus the current architecture is not suitable for extension. Applying the stylebase leads to the same conclusion.

With the help of Q-Stylebase, the quality properties of the existing architecture can be discerned without evaluating the model manually. As stated, DiSeP applies the Layers architectural pattern. Browsing the stylebase for Layers reveals that the current architecture promotes modifiability, portability and reusability, while extensibility is not promoted. Thus the Layers approach is not an optimal solution for the DiSeP architecture.

A more suitable architecture would provide loose coupling between the components, thus enabling to add new components without affecting the others. As a basic assumption the control topology should take the form of a star.

In order to resolve more suitable architecture for DiSep with special regard for extensibility, we follow the steps defined above. First, the abstraction level is switched to the conceptual, i.e. architectural pattern. Second, the quality attribute "Extensibility" is selected from the quality attribute list. Third, browsing the pattern name list reveals a list of architectural patterns promoting extensibility.

When a pattern is selected from the list, the quality attribute list is updated with the other qualities promoted by the pattern. As the designing of architectures always involves making compromises, in this stage the architect can weight the other qualities and anti-qualities promoted by the different patterns and choose the most suitable ones for closer examination on the basis of the quality requirements. However, in this simplified example they are not further considered.

Browsing the pattern name list reveals that Blackboard is one of the patterns promoting extensibility and therefore it is selected. In addition to extensibility, Blackboard promotes availability, modifiability, reliability and reusability. At the moment, no anti-qualities are set for the Blackboard in the stylebase. As no other quality requirements have been set, Blackboard seems to be a good candidate for a new architecture of the DiSeP.

Prior to starting the redesigning process for the architecture of the DiSeP, the architect pushes the "Advanced" button in order to get more details on how to apply the Blackboard pattern. As the Layers and the Blackboard architectural patterns share the same purpose, it is likely that the redesign process makes sense. By contrast, redesigning a system, implemented in Model-View-Controller pattern (Interactive systems), into Microkernel (Adaptable systems) [15] would not make sense, as the patterns are not intended for the same problem area.

The control and data topology of Blackboard reveal the mechanism through which extensibility is achieved. Adding new computation components in the role of source will not have any direct impact on other computation components. However, if the intention is to add new data (blackboard) or control components (control), Blackboard is not an appropriate solution.

If the architect is satisfied with other pattern parameters, the pattern is ready to be applied. The guide provides all the information required for applying the pattern if the pattern is not previously known.

When the necessary knowledge about the roles and types of the components, included in DiSeP, is available, new architecture can be constructed by following the Blackboard construction guide. Component types are not changed in the redesign process, which makes it is easy to discern the roles of the new components. A new architecture is then achieved through building a new connector topology and rearranging the components. Figure 6 presents architecture of the DiSeP with the Blackboard architectural pattern applied to it. At the centre of the model, the control component, Activator service, controls the other (computation and Blackboard) components. The Data storage in the role of Blackboard can be

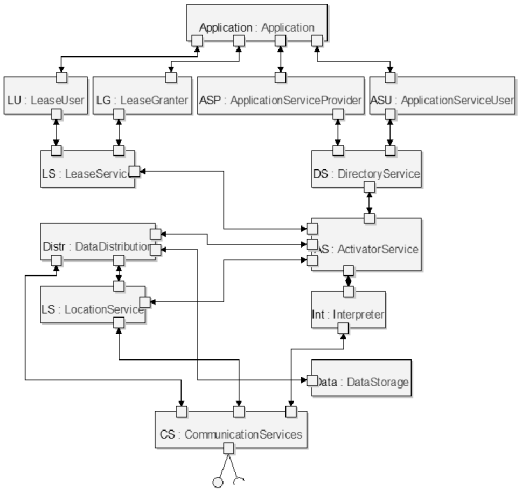

Fig. 5. DiSeP – The Layers model.

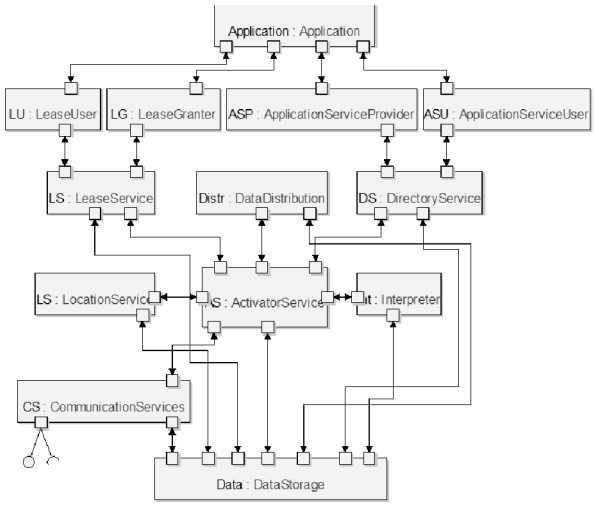

Fig. 6. DiSeP – The Blackboard model.

found communicating by data signals at the bottom of the composite structure diagram. The control topology takes the form of a star and the data topology is clearly hierarchical.

As a result of redesigning the architecture, the DiSeP becomes extendable. Adding new service components does not affect other components as all communication passes through a single centralized control component, which allows a loose coupling between the other service components to be achieved. Only minor refinements may be required to the Activator service and to Data storage when the system is extended.

6. DISCUSSION

Describing the informal aspects of patterns explicitly is not straightforward, especially in terms of quality. Currently, there is no systematic or explicit method available for defining quality attributes for the patterns. In fact, even neither a common vocabulary nor congruent definitions concerning the qualities exist.

As there are no systematic methods available for combining quality attributes and patterns, quality issues have to be defined on the basis of experience. Further questions also arise as to whether the quality attributes should be divided into more nuanced representations and if some kind of grading should be used for expressing the impact of quality more rigorously.

It is difficult to explicitly describe the qualities, promoted by the patterns, because the context affects the quality. For instance, a pattern may promote extensibility while at the same time adding more and more components into the system will weaken the performance when this was not an issue with fewer components. It is difficult to set a threshold for when to define qualities and anti-qualities for the patterns.

One may also argue the impact of a pattern on an overall quality of a system. Given that a system has been implemented with a certain architectural pattern, it should be possible to draw a conclusion about the architectural qualities but this does not, however, give any clear indications about the impact on quality when design patterns are applied.

In the case of design patterns, a viable approach could be to consider the qualities as enablers or an operator, i.e. a design pattern makes it possible to do something. For example, the Bridge design pattern promotes integrability among other things. Thus the Bridge makes it possible to integrate separately developed components with the existing ones. Although the quality of applying Bridge shows in that specific hot spot, the question arises whether this can be generalized to indicate its impact on overall quality. Using Bridge managing to implement a completely new variation point, it is clear that the overall integrability of the system is improved. But if Bridge were applied in an insignificant spot, would the overall quality really be improved?

The data model used for describing architectural patterns does not suit well to design patterns: neither control nor data topology can be easily discerned and neither is a component type expressive enough. Considering this, the framework for describing informal aspects of the design patterns explicitly requires a refined version of the data model.

In the future, we would, in addition to describing design patterns more explicitly, like to extend the Q-Stylebase with the feature of pattern recognition. This would allow the architect, by pressing a button, to see what patterns are applied in a model and also to see from what quality perspective the architecture has been designed. We are also interested in automating the redesign process, which has to be performed if the quality requirements of the architecture vary. Automating the redesign process requires support for generating patterns for the (architecture) models, and therefore, pattern generation is also under consideration.

7. CONCLUSIONS

Applying MDE in software development may decrease development costs and produce better software. However, applying MDE alone does not guarantee more efficient models; quality issues need to be emphasized in architecture development.

Our contribution is a tooling environment that brings quality-driven and model-driven engineering closer to each other. Q-Stylebase is a tool that provides a stylebase for the architect for designing, evaluating and producing appropriate architectural models. We have shown that applying the Q-Stylebase in software architecture modelling reduces the need of evaluating the quality of the architecture models manually. In addition, we have shown that Q-Stylebase helps the architect in choosing the best architectural solution in terms of quality and in redesigning existing architecture.

In conclusion, we believe that the course of action taken with Q-Stylebase is a step towards bridging the gap between quality issues and model-driven engineering.

ACKNOWLEDGEMENT

This paper was carried out at VTT within the ITEA project ip02009, FAMILIES part within the Eureka Σ! 2023 Programme.

REFERENCES

1. Niemelä, E., Matinlassi, M. and Taulavuori, A. Practical evaluation of software product family architectures. *Lecture Notes in Comput. Sci.*, 2004, **3154**, 130–145.
2. Miller, J. and Mukerji, J. *MDA, Guide, Version 1.0.1*. OMG document/2003-16-1, 2003.
3. Merilinna, J. and Matinlassi, M. Evaluation of UML tools for model-driven architecture. In *11th Nordic Workshop on Programming and Software Development Tools and Techniques NWPER'2004*. Turku, Finland, 2004, 155–163.
4. *IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems, Std-1417-2000.* IEEE, New York, 2000.
5. Kruchten, P. The 4+1 view model of architecture. *IEEE Software*, 1995, **12**, 42–50.
6. Hofmeister, C., Nord, R. and Soni, D. *Applied Software Architecture.* Addison-Wesley, Reading, MA, 2000.
7. Obbink, H., Muller, J., America, P., van Ommering, R., Muller, G., van der Sterren, W. and Wijnstra, J. G. *COPA: A Component-Oriented Platform Architecting Method for Families of Software-Intensive Electronic Products*. SPLC1, Denver, 2000.
8. Purhonen, A., Niemelä, E. and Matinlassi, M. Viewpoints of DSP software and service architectures. *J. Syst. Softw*., 2004, **69**, 57–73.
9. Dobrica, L. and Niemelä, E. Using UML notation extensions to model variability in product-line architectures. In *International Workshop on Software Variability Management ICSE'03*. Portland, Oregon, 2003, 8–13.
10. Bass, L., Clements, P. and Kazman, R. *Software Architecture in Practice.* Addison-Wesley, Reading, Massachusetts, 1998.

11. Bosch, J. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. Addison-Wesley, Harlow, 2000.
12. Matinlassi, M. and Niemelä, E. The impact of maintainability on component-based software systems. In *29th Euromicro Conference EUROMICRO'03*. Turkey, 2003, 25–32.
13. *Software Engineering – Product Quality. Part 1: Quality Model, ISO/IEC 9126-1:2001*. International Organization of Standardization and International Electrotechnical Commission, 2001.
14. Dobrica, L. and Niemelä, E. A survey on software architecture analysis methods. *IEEE Trans. Softw. Eng*., 2002, **28**, 638–653.
15. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. *Pattern-oriented Software Architecture – a System of Patterns*. Wiley, Chichester, New York, 1996.
16. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, Addison Wesley, 1994.
17. Matinlassi, M., Niemelä, E. and Dobrica, L. *Quality-driven Architecture Design and Analysis Method: A Revolutionary Initiation Approach to a Product Line Architecture*. VTT, Oulu, 2002.
18. Niemelä, E., Kalaoja, J. and Lago, P. Towards an architectural knowledge base for wireless service engineering. *IEEE Trans. Softw. Eng*., 2005, **31**, 361–379.
19. Eden, A. H. and Hirshfeld, Y. Principles in formal specification object-oriented design and architecture. In *Conference of the Centre for Advanced Studies and Collaborative Research*. Toronto, Orlando, Canada, 2001.
20. Erikson, H., Penker, M., Lyons, B. and Fado, D. *UML 2 Toolkit*. OMG Press, Wiley, 2004.
21. Shaw, M. and Clements, P. Field guide to boxology: preliminary classification of architectural styles for software systems. In *21st Annual International Computer Software & Application Conference COMPSAC'97*. Washington, DC, 1997. IEEE, Los Alamitos, CA, 1997, 6–13.
22. MySQL AB. (15.02.2005) MySQL. http://www.mysql.com/
23. MySQL AB. (15.2.2005) MySQL: MySQL Open Source License. http://www.mysql.com/company/legal/licensing/opensource-license.html

Stiilibaas kui kvaliteetjuhitava tarkvaraarhitektuuri modelleerimise vahend

Janne Merilinna ja Eila Niemelä

Mudeljuhitava tarkvaratehnika (MDE – *model-driven engineering*) peamiseks eesmärgiks on kvaliteetse tarkvara tootmisel mudeleid kasutades parandada tarkvara kvaliteeti ja vähendada tarkvaraarenduse kulusid. Mudeljuhitavas tarkvaratehnikas esitatakse tarkvaralahendused teatud mudelite abil, arendades ja hinnates lähtekoodi asemele arendatava tarkvarasüsteemi mudeleid. Nii tarkvara kvaliteet kui ka tarkvaramudelite kvaliteet sõltub süsteemi või süsteemide klassi kvaliteedinäitajate kaasamisest tarkvaraarhitektuuri arendusse. Artiklis on esitatud tarkvara arhitektuursete mudelite ja projekteerimisvõtete nn stiilibaas. Stiilibaas on integreeritud tarkvaradisaineri töökeskkonda, toetamaks tarkvaraarhitektuuri projekteerimist ja hindamaks süsteemi arhitektuurse mudeli kvaliteeti. Stiilibaasi saab kasutada ka hajusa heterogeense arenduskeskkonna sõltumatu komponendina.