Towards Systematic Integration of Quality Requirements into Software Architecture [*]

Azadeh Alebrahim, Denis Hatebur, and Maritta Heisel

University Duisburg-Essen, Germany, {azadeh.alebrahim, denis.hatebur, maritta.heisel}@uni-duisburg-essen.de

Abstract. We present a model- and pattern-based approach that allows software engineers to take quality requirements into account right from the beginning of the software development process. The approach comprises requirements analysis as well as the software architecture design, in which quality requirements are reflected explicitly.

1 Introduction

Taking quality (or non-functional) requirements into account when developing a software architecture is a demanding task, for which satisfactory solutions are still sought for. In this paper, we want to contribute to improve this situation. We present a model- and pattern-based approach for architectural design that explicitly takes quality requirements (in particular, security and performance requirements) into account.
As a basis for requirements analysis, we use Jackson's problem frame approach [6]. We have carried over problem frames to UML [11] by defining a specific UML profile, and we have implemented a tool, called *UML4PF* [1] supporting requirements analysis and architectural design based on problem frames [4]. As a basis for architectural design, we use an method that we developed for deriving architectures based on functional requirements [2].
In the present paper, we extend our previous requirements analysis and architectural design methods by explicitly taking into account quality requirements. The analysis documents are extended by quality requirements that complement functional ones. For this purpose, we have extended the UML profile [5]. The so enhanced problem descriptions form the starting point for architectural design. To design the architecture, we apply appropriate security or performance patterns and mechanisms and define quality stereotypes that serve as hints for implementers.
The rest of the paper is organized as follows. We present the basics on which our approach builds in Sect. 2, namely problem frames and security and performance patterns and mechanisms. In Sect. 3, we present the UML profile we defined to carry over the problem frame approach to UML. Section 4 is devoted to describing our approach in more detail. Related work is discussed in Sect. 5, and conclusions and future work are given in Sect. 6.

[*]Part of this work is funded by the German Research Foundation (Deutsche Forschungsgemeinschaft - DFG) under grant number HE3322/4-1.

[1]Available under http://swe.uni-duisburg-essen.de/en/research/tool/

2 Basic Concepts

In this section, we introduce the basic concepts our approach relies on.

2.1 Requirements Description using Problem Frames

Problem frames are patterns to describe software development problems. They were proposed by Michael Jackson [6]. A problem frame basically consists of *domains*, *interfaces* between them, and a *requirement*. The task is to construct a *machine* (i.e., software) that improves the behavior of the environment (in which it is integrated) in accordance with the requirements.
Software development with problem frames proceeds as follows: first the environment in which the machine will operate is represented by a *context diagram*. A context diagram consists of machines, domains and interfaces. Then, the problem is decomposed into subproblems, which are represented by *problem diagrams* that can be instances of problem frames. A problem diagram consists of a submachine of the machine given in the context diagram, the relevant domains, the interfaces between these domains, and a requirement. Figure 1 shows a problem diagram in UML notation.

2.2 Mechanisms and Patterns for Performance and Security

To satisfy performance and security requirements, different mechanisms – also called patterns – are available [3, 10]. *Load Balancing* is such a mechanism that is used to distribute computational load evenly over two or more hardware components. The load balancing pattern consists of a component, which is called *Load Balancer*, and multiple hardware components that implement the same functionality. *Encryption* is an important means to achieve confidentiality. A plaintext is encrypted using a secret *key* and decrypted either using the same key (symmetric encryption) or a different key (asymmetric encryption).

3 Requirements Engineering

It is important that the results of the requirements analysis with problem frames can be easily re-used in later phases of the development process. Since UML is a widely used notation to express analysis and design artifacts in a software development process, we defined a new UML profile [4, 2] that extends the UML meta-model to support problem-frame-based requirements analysis with UML. This profile can be used to create the diagrams for the problem frame approach. To address quality requirements in the requirement engineering process we enhance our UML profile with annotations for quality requirements as stereotypes.

3.1 UML Profile for Problem Frames

Using specialized stereotypes, our UML profile allows us to express the different diagrams occurring in the problem frame approach using UML diagrams.
A class with the stereotype ≪`machine`≫ represents the software to be developed (possibly complemented by some hardware). Jackson distinguishes the domain types biddable domains (represented by the stereotype ≪`BiddableDomain`≫) that are usually people, causal domains (represented by the stereotype ≪`CausalDomain`≫) that comply with some physical laws, and lexical domains (represented by the stereotype ≪`LexicalDomain`≫) that are data representations.
In problem diagrams, *interfaces* connect domains, and they contain *shared phenomena*. Shared phenomena may be events, operation calls, messages, and the

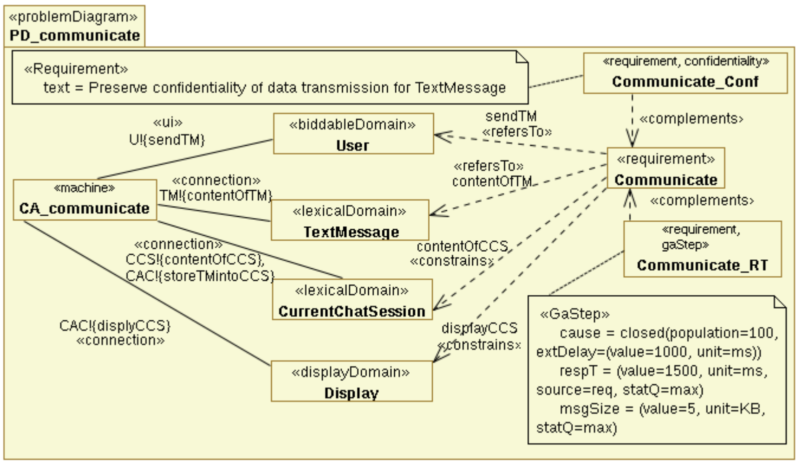

Fig. 1. Problem diagram for the requirement *Communicate*

like. They are observable by at least two domains, but controlled by only one domain, as indicated by an exclamation mark. For example, in Fig. 1 the notation *U!sendTM* (between *CA_communicate* and *User*) means that the phenomenon *sendTM* is controlled by the domain *User*. The interfaces are marked with specializations of the stereotype ≪`connection`≫, e.g., a user interface (≪`ui`≫) between *User* and *CA_communicate* machine in Fig. 1.
The stereotype ≪`requirement`≫ represents a functional or quality requirement. When we state a requirement we want to change something in the world with the machine to be developed. Therefore, each requirement constrains at least one domain. This is expressed by a dependency from the requirement to a domain with the stereotype ≪`constrains`≫. A requirement may refer to several domains in the environment of the machine. This is expressed by a dependency from the requirement to a domain with the stereotype ≪`refersTo`≫.
The problem diagram in Fig. 1 considers a chat application introduced in more detail in Sect. 4. It describes the requirement *Communicate*, e.g., it states that the *CA_communicate* machine can show to the *User* the *CurrentChatSession* on its *Display* (*CAC!*{*displayCCS*}). The requirement constrains the *CurrentChatSession* of the *User* and its *Display*. The requirement refers to the users and the text messages.
The problem frame approach substantially supports developers in analyzing problems to be solved. It points out what domains have to be considered, and what knowledge must be described and reasoned about when analyzing a problem in depth. Developers must elicit, examine, and describe the relevant properties of each domain. These descriptions form the *domain knowledge* are specified in *domain knowledge diagrams*.

3.2 Annotating Problem Descriptions with Quality Requirements

The problem frame approach proposed by Jackson provides a method that addresses functional requirements only. Quality requirements are not considered. We extended our UML profile for problem frames to complement functional requirements with security requirements [5]. Classes with stereotypes such as ≪`confidentiality`≫, ≪`integrity`≫ and corresponding attributes such as *attacker* or *stakeholder* address security requirements. The dependency from a quality requirement to a requirement is expressed with the stereotype ≪`complements`≫ (see Fig. 1). To provide support for annotating problem descriptions with performance requirements, we use the UML profile MARTE (Modeling and Analysis of Real-time and Embedded Systems) [12]. We focused on the GQAM package (Generic Quantitative Analysis Modeling) that contains basic concepts for modeling and analysis of domains based on software behavior, in particular performance. To define workload and behavior concerns we make use of the GQAM_Workload package by instantiating the appropriate attributes of this package. Each *BehaviorScenario* is composed of *Steps*, each of which can be refined as another *BehaviorScenario.* A behavior scenario is triggered by the *WorkloadEvent*, which may be generated by a stated *ArrivalPattern* such as the *ClosedPattern* that allows us to model a number of concurrent user and a think time (the time that the user waits between two requests) by instantiating the attributes *population* and *extDelay.* We define a *BehaviorScenario* composed of one *Step* for the requirement *Communicate_RT* (see Sect. 4.2), which is refined in three *BehaviorScenario* instances, each of which is composed of a single *Step.* The *Step* instances represent the requirements *Send_RT*, *Forward_RT* and *Receive_RT* that stand in the precedence relationship *Sequence* [12, p. 289].

4 Deriving quality-based Architectures

We now present our approach to derive software architectures, taking quality requirements into account. It comprises requirements analysis as well as the software architecture design. We illustrate our approach by a chat application, which allows a text-message-based communication via private I/O devices. Users should be able to communicate with other chat participants in a same chat room. We consider the *Communicate* functional requirement with the description "*Users can send text messages to a chat room, which should be shown to the users in that chat room in the current chat session in the correct temporal order on their displays*" and its corresponding quality requirement *Response Time* with the description "*The sent text message should be shown on the receiver's display in 1500 ms maximum*". Moreover, *Confidentiality* of the text messages should be preserved. Note that in order to specify performance and confidentiality requirements properly, more details have to be given.

4.1 Problem Diagrams

As described in Sect. 2.1, the first step in the software development process based on problem frames is to create a context diagram (not shown). We decompose the overall problem into subproblems represented by problem diagrams. Each problem diagram describes one subproblem with the corresponding requirement. We focus on the requirement *Communicate.* The corresponding problem diagram using our UML profile for problem frames is depicted in Fig. 1. It consists of the domains *User, TextMessage, CurrentChatSession* and *Display.* The requirement *Communicate* refers to the domains *User* and *TextMessage*, expressed by the stereotype ≪`refersTo`≫ and constrains the domains *CurrentChatSession* and *Display*, expressed by the stereotype ≪`constrains`≫.

4.2 Annotate Problem Diagrams with Quality Requirements

In this step, we address quality requirements by annotating problem diagrams with suitable stereotypes. The requirement *Communicate* is complemented by the confidentiality requirement *Communicate_Conf* that requires confidentiality of data transmission for *TextMessage* and the response time requirement *Communicate_RT* representing one *BehaviorScenario* composed of one *Step* described with the stereotype ≪`gaStep`≫ (see Fig. 1). The response time requirement is modeled by instantiating the relevant attributes of the *Step* class in the MARTE GQAM_Workload package described in Sect. 3.2. The *cause* attribute represents the triggered event, which is in our case a *ClosedPattern* with 100 concurrent users (*population*), each of which needs a think time of 1000 ms (*extDelay*). The *respT* attribute states that the required response time for sending text messages should be 1500 ms maximum. The *msgSize* attribute states that the sending text messages should be 5 KB maximum.

4.3 Choose Design Alternative and Create Architecture

We first create an initial architecture, where each machine domain in a problem diagram is mapped to a component. The initial architecture for the chat application (not shown) contains – among others – a component *CA_communicate* corresponding to the machine domain *CA_communicate* of Fig. 1.
The software architect then needs to take a design decision concerning the kind of distribution, e.g., client-server, peer-to-peer, or standalone. In the following, we describe the approach for a client-server architecture in more detail.
After having chosen a client-server architecture, we go back to the requirements description and **split the problem diagrams** in such a way that each subproblem is allocated to only one of the distributed components. This may lead us to introduce connection domains[2], e.g., networks. In our example, the problem diagram depicted in Fig. 1 is split into three problem diagrams, which address the problems of sending text messages to the server that belongs to the client (*Send*), forwarding text messages from the server to the receivers that belongs to the server (*Forward*), and receiving text messages that belongs to the client (*Receive*). For each of these three subproblems, we introduced the connection domain *Network* to achieve the distribution.
Analogously to splitting the problem diagrams, we also have to **split the corresponding quality requirements**. In case of a response time requirement, the response time should be divided so that all subproblems together satisfy the desired response time. The *Communicate* requirement states a response time of 1500 ms maximum. This must be achieved through the three subproblems *Send, Forward, Receive* and the time for data transmission over the network. We cannot meet the performance and specifically response time requirements, if we have no knowledge about the real circumstances in the environment. Therefore we specify knowledge about the network and the computational power of clients and server in a domain knowledge diagram. It contains specific knowledge about client and server, e.g., the number of processor cores, processor speed and memory. Additionally, we assume that the response time to transmit data over a network with 64 kb/s minimum is 400 ms.
To fulfill the confidentiality requirement for the problem *PD_communicate* (Fig. 1), we require confidentiality for each subproblem. Therefore, we annotate each subproblem with a corresponding refined confidentiality requirement. This requirement contains a *stakeholder* that is interested in preserving the confidentiality of data, and an *attacker* that the chat application should be protected against.

[2]These are domains needed to establish a connection between other domains [6].

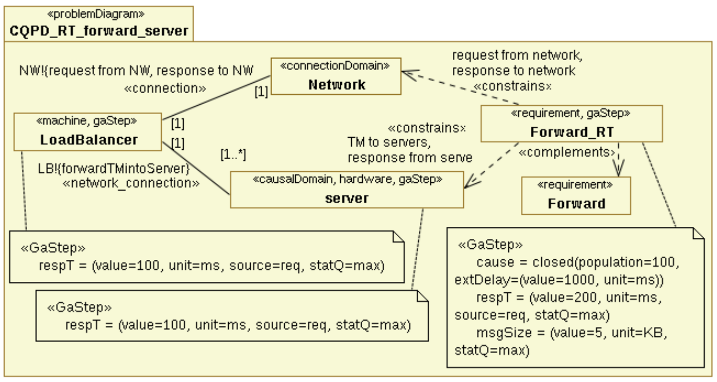

Fig. 2. Concretized quality problem diagram for the quality requirement *Forward_RT*

The stakeholder in our case is the *User*, and the attacker is a *NetworkAttacker* who is able to attack the data transported over the network.

Concretized Quality Problem Diagrams describe solution approaches in terms of mechanisms and patterns. We elaborate the problem diagrams annotated with quality requirements from the previous step by introducing domains reflecting specific solution approaches.

For example, the problem diagram for the *Send* problem describes the problem of sending text messages with two additional quality requirements for security and performance, respectively. The requirement for performance states that sending a text message should be performed within 200 ms (allocated part of the 1500 ms). However, this requirement cannot be achieved by architectural means. Instead, it must be taken care of in the implementation. In such a case, we annotate the corresponding machine with a stereotype that serves as a hint to develop a particularly efficient implementation (≪`gaStep`≫) or an implementation that does not leak information (≪`confidentiality`≫).

The security requirement describes that a text message should be transmitted confidentially over an insecure network. To take this quality requirement into account, we specify the concretized quality problem diagram including an *Encryption* machine and domains for keys used for asymmetric encryption. This decision necessitates to also introduce new components on the receiver side, namely a new machine *Decryption* and a domain *ReceiverUserPrivateKey*.

In order to address the response time requirement in the *Forward* problem even under high load, we introduce a new machine *LoadBalancer* (see Fig. 2). It distributes the load from the network across several server components.

By now we have provided a suitable basis for quality-aware architectural design in the requirements analysis phase. To **design an architecture** that achieves the required level of performance and security, we make use of the split problem diagrams to allocate components to the client and to the server. Each machine in the split problem diagrams belongs to a component in the client or in the server according to functionality of that submachine. To design the architecture, we merge related components, apply design patterns (e.g., Facades), and use the solution domains for quality requirements (e.g.,*LoadBalancer*). The resulting software architecture for the chat application – represented as a UML composite structure diagram – is shown in Fig. 3.

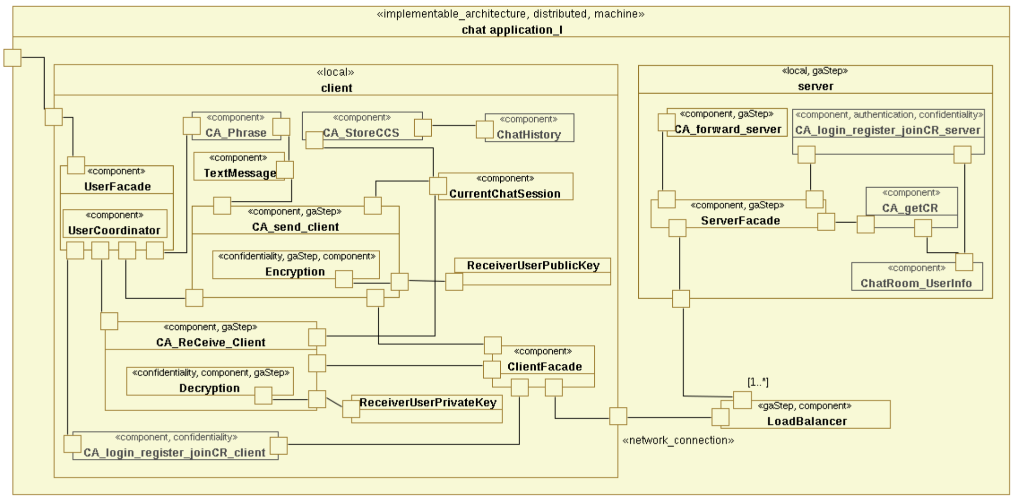

Fig. 3. Client-server architecture for the chat application

5 Related Work

Previous work often considers only one type of quality requirements during the software development process, e.g., security.
An approach to transform security requirements to design is provided by Mouratidis and Jürjens [9]. It starts with the goal-oriented security requirements engineering approach Secure Tropos [8], and connects it with a model-based security engineering approach, namely UMLsec [7].
Yskout et al. [14] present a semi-automated approach to support the transition from security requirements to architecture. They focus on delegation, authorization and auditing as security requirements. They presuppose an architecture that fulfills the functional requirements, and they apply security solutions to the functional architecture by transforming security requirements.
Attribute Driven Design (ADD) [13] is a method to design a conceptual architecture. It focuses on the high-level design of an architecture, and hence does not support detailed design. Identifying mechanisms to achieve quality attributes relies on the architect's expertise.
Q-ImPrESS [1] is a project that focuses on the generation and evaluation of architectures according to quality properties, in particular performance. The phases design and implementation of the software development process are particularly in focus. In contrast to our contribution, it does not use requirements descriptions as a starting point.

6 Conclusion

In this paper, we have presented a UML-based approach to design software architectures from requirements, taking quality requirements into account. We provide means to specify quality requirements thoroughly with problem diagrams,

and we incorporate mechanisms or patterns addressing these requirements explicitly in the software architecture.
Our approach builds on established techniques such as problem frames, security and performance patterns. Its novelty lies in the fact that the different approaches are integrated and intertwined explicitly by an underlying methodology and a common notation. The notation as well as the methodology are open and can be developed further to enhance the power and breadth of the approach. In the present work, we have not investigated possible conflicts between different quality requirements. We strive for a more systematic treatment of conflicting quality requirements. Moreover, we have concentrated on structural descriptions of software architectures. In the future, we will extend our approach to also support deriving behavioral descriptions for the developed architectures and automatically checking their coherence with the structural descriptions.

References

1. S. Becker, S. Dešić, J. Doppelhamer, D. Huljenić, H. Koziolek, E. Kruse, M. Masetti, W. Safonov, I. Skuliber, J. Stammel, M. Trifu, J. Tysiak, and R. Weiss. Q-ImPrESS Project Deliverable D1.1 – Requirements document. final version, Q-ImPrESS Consortium, 2009.
2. C. Choppy, D. Hatebur, and M. Heisel. Systematic architectural design based on problem patterns. In P. Avgeriou, J. Grundy, J. Hall, P. Lago, and I. Mistrik, editors, *Relating Software Requirements and Architectures*, chapter 9. Springer, 2011. To appear.
3. C. Ford, I. Gileadi, S. Purba, and M. Moerman. *Patterns for Performance and Operability*. Auerbach Publications, 2008.
4. D. Hatebur and M. Heisel. Making Pattern- and Model-Based Software Development More Rigorous. In J. S. Dong and H. Zhu, editors, *Proc. of 12th Int. Conf. on Formal Engineering Methods*, LNCS 6447, pages 253–269. Springer, 2010.
5. D. Hatebur and M. Heisel. A UML profile for requirements analysis of dependable software. In E. Schoitsch, editor, *Proc. of the Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP)*, LNCS 6351, pages 317–331. Springer, 2010.
6. M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.
7. J. Jürjens. *Secure Systems Development with UML*. Springer, 2005.
8. H. Mouratidis. *A Security Oriented Approach in the Development of Multiagent Systems: Applied to the Management of the Health and Social Care Needs of Older People in England*. PhD thesis, University of Sheffield, U.K., 2004.
9. H. Mouratidis and J. Jürjens. From goal-driven security requirements engineering to secure design. *Int. J. Intell. Syst.*, 25:813–840, 2010.
10. M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. Wiley & Sons, 2005.
11. "UML Revision Task Force". *OMG Unified Modeling Language (UML), Superstructure*. http://www.omg.org/spec/UML/2.3/Superstructure/PDF.
12. "UML Revision Task Force". *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. http://www.omg.org/spec/MARTE/1.0/PDF.
13. R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord, and B. Wood. Attribute-Driven Design (ADD). Version 2.0, Software Engineering Institute, 2006.
14. K. Yskout, R. Scandariato, B. D. Win, and W. Joosen. Transforming security requirements into architecture. In *Proc. of the 3rd Int. Conf. on Availability, Reliability and Security*, pages 1421–1428, USA, 2008. IEEE Computer Society.