Automated Decision Support for Recurring Design Decisions Considering Non-Functional Requirements

Axel Busch
Karlsruhe Institute of Technology
busch@kit.edu

Abstract: Planning high quality software means more than regarding functionality. Considering non-functional requirements, implementing them and understanding their effects on the software architecture remain often an open question. Therefore, in this paper, we present an approach that provides decision support in a software development process for recurring design decisions in the field of non-functional requirements. The approach defines a design decision model that allows to encapsulate the reasoning of design decisions, make them reusable and use them to enable automated feedback in a decision making process. At the end, this approach increases the developer's productivity by reusing design decisions and therefore allows to implement requirements with lower overhead and to improve the architecture quality by a tool assisted decision support process.

1 Motivation and Introduction

Despite of many improvements in software engineering processes, in recent years even high budget software projects delayed or even failed. The increasing size and complexity of these software projects requires an explicit consideration of quality attributes. Otherwise, the software will not fulfill the stakeholders expectations in terms of non-functional attributes like security, usability or reliability. Considering quality attributes in a software architecture design is one major task when making architecture design desicions. Due to missing quantification methodologies for many considered quality attributes, the influence of a decision on other quality attributes remains unclear. Therefore, it is often difficult to estimate in advance which attributes an architectural decision would affect. This means that if a developer has made a decision that improves one specific attribute, the influence on other quality attributes, e.g. performance, remains an open question.

Our approach addresses these issues when considering recurring design decisions, following the concept of the component-based software engineering (CBSE) paradigm. Recurring design decisions are decisions that are applicable in many projects and organizations. Examples for recurring design decisions may be introducing an Intrusion Detection System or selecting a messaging middleware. In our approach, we encapsulate the reasoning of such a recurring design decision in one entity to make it reusable and model the effects on the quality attributes on a software architecture to allow automated requirements trade-off decisions.

A design decision comprises a generic part that is architecture independent (e.g. component interaction) and a part that is specific for a particular architecture (e.g. concrete components). Such a design decision could be implemented in different ways by different components to be used and different deployment configurations of these components. Each configuration may have its own quality attributes, e.g. different levels of security or different performance properties. These characteristics of design decisions could be encapsulated to one entity, a design decision model entity. This approach provides a design decision repository that contains these entities representing solutions for recurring design decisions, in order to fulfill certain quality requirements. An automated decision support system uses these design decision model entities to support developers at trade-off decisions to select the best suitable solution in an existing architecture according to the project's non-functional requirements.

2 Related Work

Svahnberg et al. showed in [SW05] a six-step decision support approach to evaluate different architecture candidates according to the considered quality requirements using the multi-criteria decision method Analytic Hierarchy Process (AHP). The approach needs a fully manual definition of the effects on the quality attributes for each design alternative. Kazman et al. propose in [KKCC00] their Architecture Tradeoff Analysis Method (ATAM) that considers the architecture's degree of fulfilment of quality requirements and how they interact and influence each other. ATAM allows to think systematically about quality requirements and the impact of architecture design decisions on quality attributes of the system. The resulting questions are informally specified and are therefore not convenient to be used in automated processes. Manteuffel et al. developed in [MTK$^+$14] an add-in for Sparx Systems' Enterprise Architect allowing to model and document architecture decisions. Their add-in allows to document the link between a design decision and other model elements. Their approach focuses on documenting design decisions in software architectures, but omits capturing the effects on the quality attributes or mechanisms to use the results in a decision support process. Horcas et al. showed in [HPF14] a Software Product Line approach to model functional quality attributes separated from the base software architecture to modularize them and to make them reusable. They use the Common Variability Language to inject elements into the architecture. The approach focuses on variability but does not consider trade-off decisions or providing feedback for architecture improvements.

3 Goals and Questions

The proposed approach aims to provide automated feedback in a decision making process allowing to select the best suitable solution for recurring design decisions when implementing non-functional requirements. It includes a design decision repository with design decision model entities to be used in an automated process to support a user to select architecture design decisions that are best suiting for the project's non-functional requirements.

The following goals lead the development of the main goal: *G1: Encapsulate and reuse of design decisions to be implemented in a late architecture design process. G2: Exploring design decision degrees of freedom to give tool-supported feedback for requirements-driven architecture adaptation, requirements prioritization and refinement.*

The following research questions will be answered to reach these goals: *RQ 1: How to represent design decisions and their degrees of freedom to be used in an automated decision support process? RQ 2: How to use such a design decision representation in a feedback process for requirements and architecture improvements?*

These questions are divided in the subquestions as follows: *RQ 1.1: Which attributes of non-functional requirements have to be modeled in which way? RQ 1.2: Which measure is adequate for non-quantifiable requirements for the purpose of an automated decision support? RQ 1.3: How can a general model to represent design decisions look like? How can architecture generic and architecture specific parts be described and resolved? RQ 1.4: How can the influence of a design decision on the quality attributes be represented? RQ 1.5: Which design decision degrees of freedom can be derived and how can they be represented in a formalized model to be used for an automated design space exploration? RQ 2.1: How to set up a feedback process to support selecting and implementing suitable solutions in the software architecture? RQ 2.2: How to extend the feedback process to refine and prioritize the list of requirements? RQ 2.3: What tool-to-user interaction is needed to support the feedback process?*

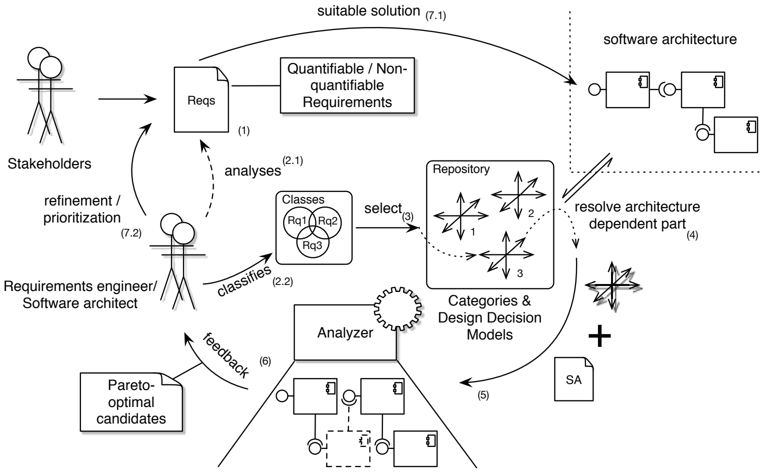
suitable solution (7.1)
software architecture
Reqs
Quantifiable / Non-quantifiable Requirements
(1)
Stakeholders
analyses (2.1)
refinement / prioritization
(7.2)
Classes
Rq1
Rq2
Rq3
select (3)
Repository
1
2
3
resolve architecture dependent part (4)
Requirements engineer/ Software architect
classifies (2.2)
Categories & Design Decision Models
Analyzer
feedback
(6)
Pareto-optimal candidates
SA
(5)

Figure 1: Main steps of our approach

4 Approach

Figure 1 shows an overview of the approach described in this paper. A design decision support process needs to understand the requirements and implement them in the most suitable way. The approach focuses on a decision support for selecting and implementing non-functional requirements that are potentially to be solved by recurring design decisions.

Requirements elicitation typically results in an initial, not prioritized and imprecise defined list of requirements (1) that might be project relevant. The requirements engineer analyzes this first sketch of requirements (2.1) and classifies (2.2) them into main categories, e.g. security or usability. Then he selects (3) for each requirement the potentially suitable design decision model entities from the repository or adds a newly implemented entity. Each design decision model entity contains an architecture dependent part that needs to be resolved by the software architect for the particular architecture (4). The selected models are used with the software architecture to calculate (5) pareto-optimal architecture candidates for all considered quality requirements and their properties. The feedback system of the approach can now use the resulting candidates (6) to provide the requirements engineer and the software architect pareto-optimal solutions according to their focused quality attributes as well as their achievable level of fulfillment. At this point the requirements engineer has two options: If he agrees with the achieved level of fulfillment of a solution, this is forwarded to the software architect (7.1) to be used in production. If the anticipated level of fulfillment may not be achieved, the requirements engineer can now decide to either improve the requirements definitions or to refine their prioritization (7.2). In the latter cases, the process would be iteratively performed as long as the anticipated level of fulfillment is achieved. At the end, the resulting architecture conforms to a suitable trade-off in order to fulfill the requirements suggested by the stakeholders.

5 Evaluation

Our approach will be evaluated in an empirical study. We plan to assess the benefits of the approach with respect to the following two main evaluation goals: *EG 1: Improving the quality of architecture design decisions. EG 2: Reducing the time to assess the different design alternatives to fulfill a certain requirement according to its reasoning.*

To assess our EGs, we plan to perform two quasi-experiments, one with doctoral researchers and one with graduate students. Each group will be divided into two subgroups: One experimental group that performs the experiment with tool-support and one control group that performs the task without tool-support. To evaluate our approach, we will analyze the results to identify the benefits in terms of achieved software quality and used time. We will use the architecture and documentation of an open source software project that would provide a strong basis for a representative evaluation. The project should contain a representative software architecture and documented non-functional requirements implementations. These artifacts provide us a real-world initial software architecture and software requirements to be implemented with our approach. Both groups will be asked to make architectural decisions to implement these requirements. We are interested in answering two questions to evaluate our EG 1: Can our approach improve the quality for each group compared to the control group? Can less experienced users that use our approach achieve similar quality as more experienced users without support? The goal of EG 2 is to analyze the time that a developer spends on assessing the different design alternatives to improve the non-functional requirements fulfillment. Again, we use the results of both groups to derive two kinds of results: First, we evaluate the development time grouped by the developer's experience. Second, we compare the results of the lower and higher experienced developers to analyze if the lower experienced group may close up to the higher experienced group in terms of development time and resulting quality.

References

[HPF14] Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes. Injecting Quality Attributes into SW Architectures with the Common Variability Language. CBSE '14. ACM, 2014.

[KKCC00] R. Kazman, M. Klein, P. Clements, and N. Compton. ATAM: Method for Architecture Evaluation, 2000.

[MTK+14] C. Manteuffel, D. Tofan, H. Koziolek, T. Goldschmidt, and P. Avgeriou. Industrial Implementation of a Documentation Framework for Arch. Decisions. WICSA, 2014.

[SW05] M. Svahnberg and C. Wohlin. An Investigation of a Method for Identifying a Software Arch. Cand. with Respect to Quality Attributes. *Empirical Software Engineering*, 2005.