Assessing the Quality Impact of Features in Component-Based Software Architectures

Axel Busch[1(✉)], Dominik Fuchß[1], Maximilian Eckert[2], and Anne Koziolek[1(✉)]

[1] Karlsruhe Institute of Technology, Karlsruhe, Germany
{busch,koziolek}@kit.edu, dominik.fuchss@student.kit.edu
[2] SAP Customer Experience, Munich, Germany
maximilian.eckert@sap.com

Abstract. In modern software development processes, existing software components are increasingly used to implement functionality instead of developing it from scratch. Reuse of individual components or even more complex subsystems leads to more cost-efficient development and higher quality of software. Subsystems often offer a variety of features whose use is associated with unclear effects on the quality attributes of the software architecture, such as performance. It is unclear, whether the quality requirements for the system can be met by using a certain feature of a particular subsystem. After initial selection, features must be incorporated in the target architecture. Due to a multitude of possibilities of placing the subsystem in the target system to be used, many architectural candidates may result which have to be evaluated in existing decision support solutions. The approach presented here enables software architects to automatically evaluate with the help of software architecture models the effects on quality of using individual features in an existing software architecture. The result helps to automatically evaluate design decisions regarding features and to decide whether their use is compatible with the quality requirements. We show the benefits of our approach using different decision scenarios driven by features and their placement alternatives. All scenarios are automatically evaluated, demonstrating how decisions can be made to best meet the requirements.

Keywords: Automated design decision optimization · Quality impact of features · CBSE

1 Introduction

Modern software systems support an increasing number of functionalities. The influence of the software architecture on the subsequently attainable software quality has been shown to be one of the critical factors. Therefore, it is important to consider quality attributes at design time. A subsequent change of the software architecture to implement certain functionalities without considering

T. Bures et al. (Eds.): ECSA 2019, LNCS 11681, pp. 211–219, 2019.
https://doi.org/10.1007/978-3-030-29983-5_14

the quality properties in advance can easily lead to high refactoring costs. For this reason, software architects want to evaluate their design decisions regarding the software architecture at an early stage. In particular, use of the paradigm of component-based software design has shown that there already are approaches that produce very promising results in predicting quality properties during the design phase, an example being the Palladio approach [12]. Such approaches benefit from modern software development, in which most of the functionalities are not longer developed from scratch, but are often reused in the form of libraries or subsystems. Such libraries often provide many features, i.e. function compositions that fulfill concerns. By reusing libraries or subsystems, not only the pure functionalities or the features are reused, but also their quality. Using such software artifacts reduces development time and the risk of recurring, already solved errors in a new development. Nevertheless, prediction of quality attributes at design time for reusing different systems is not trivial, especially when software developers have to decide among several similar systems or solutions. In addition to the supported features, the systems also differ in their quality attributes. When making decisions, the software architect is therefore facing the task of designing the right system to meet both the functional and quality requirements. Existing approaches for supporting design decisions with the quality attributes of software architectures, such as ArcheOpteryx [1], ArcheE [4], and PerOpteryx [11], already allow for an automatic exploration of architecture candidates with regard to different degrees of freedom. However, none of the approaches mentioned above provides decision support for evaluating the impact of using particular features on the quality attributes of the overall system. Nor do the approaches mentioned entail any recommendations which subsystem might be the best solution in order to maintain the defined quality properties.

We base on PerOpteryx and extend the approach for optimizing software architectures in the design phase or in evolution scenarios by regarding the features and the quality of solutions of the same type. This automatically supports the decision-making process of the software architect when features should be evaluated in terms of quality attributes regarding different implementations of functionally similar solutions. These extensions enable software architects to automatically analyse and optimize the effects of the implementation of functional requirements on quality attributes of the software system, such as performance, reliability, and monetary costs in the design phase. Furthermore, we can analyze whether the configuration of placement and assembly of the new features affect the quality attributes. The result of the automatic analysis and optimization helps software architects to choose the optimal solution among different functionally similar systems. This increases the efficiency of software development by reducing early wrong decisions, improves the quality of the resulting system, and reduces the risk of project delays or the failure of software projects.

2 Background

2.1 Design Space Exploration: PerOpteryx

We apply our methodology based on PerOpteryx [11], but the concepts are not limited to this approach. The PerOpteryx approach explores the huge set of software architecture configurations, in which each configuration is a specific combination of all possible design decisions. Thus, PerOpteryx supports making well-informed trade-off decisions for performance, reliability, and costs. For the design space exploration, PerOpteryx makes use of so-called *degrees of freedom* of the software architecture that can either be predefined and derived automatically from the architecture model or be modelled manually by the architect. As an example of a manually modelled degree of freedom, let us consider that some of the architecture's components offer standard functionality, for which other implementations (i.e. other components) are available. In this example, let us assume there is a available component `QuickDatabase` that can replace a `Database`. Assuming that `QuickDatabase` demands less resources but is more expensive than `Database`, the resulting architecture model has better response times but higher costs. The degrees of freedom span a design space and can be explored automatically. Together, they define a set of possible architecture models. Each of these possible architecture models is defined by choosing one design option for each degree of freedom instance (DoFI). We call such a possible architecture model a *candidate model*. The set of all possible candidate models corresponds to the set of all possible combinations of design options. We call this set of possible architecture models the *design space*.

Using the quantitative quality evaluation provided by the PCM analysis tools, PerOpteryx can determine performance, reliability, and cost metrics for each candidate model. The quality evaluation for a quality attribute can be expressed as a *quality evaluation function* from the set of valid PCM instances to the set of possible values of the quality metric. In addition to the evaluation functions, PerOpteryx requires a specification of whether a quality is to be maximized or minimized. Based on the DoFIs (as optimization variables) and the quality evaluation functions (as optimization objectives), PerOpteryx uses genetic algorithms and problem-specific heuristics to approximate the Pareto front of optimal candidates. Details on the optimization are not required for the discussion in this paper, but can be found in [9,10].

In its previous version described in this section, PerOpteryx does not support the analysis of the effect of reusing particular features of subsystems that require more complex modifications on the architecture model. The effects of using a single feature or a combination of features across the boundaries of multiple solutions cannot be studied meaningful by the previous PerOpteryx.

2.2 Feature Completion Meta Model

For the automatic evaluation of the effect of individual features when reusing components on the quality attributes of the overall system, we use the meta

model from [7]. The meta model offers entities for structuring similar systems with the same underlying features. It consists of three parts, the feature completion definition, solution definition, and transformation description. The feature completion definition part consists of a `FeatureCompletionRepository` that stores all predefined `FeatureCompletions`. Such a feature completion is an abstract entity that can be decomposed into its basic elements, namely the *Feature Completion Components* (FCC). These basic elements define the abstract architecture of a feature completion (FC) that any realizing feature completion solution such as a MySQL DB for the DBMS FC must apply. Abstraction allows the automatic integration of inhomogeneous architectures of similar solutions into a target architecture. Similar to the more concrete software components, abstract FCCs can require each other's services or offer services themselves. Additionally, we define a model for `FeatureObjectives`. This model combines features in groups. The task of these groups is to represent interchangeable or mutually exclusive features. Let us consider a DBMS example. The FC DBMS could consist of two FCCs (simplified) - the unit for reading and retrieving structured data (i.e., `StructuringDataUnit`) and the unit for actually storing these data (i.e., `DataStorageUnit`). Correspondingly, `StructuringDataUnit` would offer services that require and provide unstructured data, while `DataStorageUnit` would require and provide the unstructured data for storage purposes. All the systems of the class of DBMS on the market would then be applied to this architecture (solution definition). For this, we use annotations that identify the integration points of the completion solutions in the target software architecture. Using an inclusion mechanism, which is also provided in the meta model (transformation description), the different solutions of the same feature completion can then be automatically included into the target architecture. Given the annotated components and transformation descriptions, the integration engine determines how a feature completion solution has to be integrated into the target software architecture.

2.3 Feature Completion Integration Mechanism

We can use two different types of integration mechanisms to incorporate the appropriate features in the target system. The first one is the `Adapter-InclusionMechanism`. Whenever a connection is to be established between a component in the target system and in the solution system, a new adapter component is (automatically) generated. This adapter component requires the interface of the solution component. Furthermore, it requires and provides the interface of the target component. The provided interface of the target component is connected to the corresponding required interface of the adapter. The adapter is then connected to the solution component using the corresponding interface. For each call to the provided interface, the adapter delegates the call to the target component and an external call to the subsystem component. Afterwards, the calls to the target component and its assembly contexts, respectively, are redirected to that of the adapter. As a result, the feature is incorporated in

the system and can be used. In addition, the architect can also define the interfaces and signatures for which this mechanism should be performed. The second integration mechanism is the `BehaviorInclusionMechanism`. This mechanism allows a more fine-grained definition of how a feature should be built into the software architecture. Thus, it is possible to define that a call to the solution is to be executed in specific control structures of an RDSEFF. It is also possible to describe that at the beginning or at the end of a method call, this call is executed in the solution system.

3 Approach

Our approach consists of two parts: First, we demonstrate modelling of the features supported by the subsystem and the alignment of the features with its executing components. Second, candidates are created, evaluated, and optimized together with the target system architecture using degrees of freedom, which are spanned using the possible features and their configuration.

A subsystem provides services that can be reused in the target system. Only services that are provided by the subsystem via system external interfaces can be reused by the target system. Features will either be realized by FCCs as a whole (i.e. all provided interfaces of an FCC) or by a subset of these interfaces. In addition to system external interfaces, internal interfaces of the subsystem can also implement features. These features may be required by other features in order to implement their actual functionality. From this set of candidates, the software architect can then select the best candidate according to the project requirements.

Using the subsystem features and its associated architecture, the optimizer first generates the degree of freedom instances. If a particular solution supports a feature, it can be used and vice versa. The selection of a feature opens up further degrees of freedom, such as the position in the target system or the allocation of the solution itself. In addition, the three degrees of freedom, namely components exchange, component allocation, and development of hardware resources are included in the DoFI. In the next step, the software architecture candidate is created and integrated according to the previously generated DoFI. The evaluation required for the optimization is then performed according to the quality attributes (e.g., performance, reliability, cost) defined before. In the end, the software architect selects the best candidate from the resulting set of Pareto-optimal architecture candidates.

In order to extend the approach to decision support in software architecture design PerOpteryx by the approach described, we have adapted three parts: First, the meta model for the definition of reusable subsystems must be extended by the possibility of modelling supported features by a particular class of subsystems. Secondly, the degree of freedom model must be extended to include the existence or non-existence of features when creating architecture candidates. Depending on the solution and the features supported by this solution, the architecture candidates must be created. In addition, we need a degree of freedom

modelling of placement configurations of the feature in the target architecture. Finally, the model weaving mechanism must be extended so that the corresponding model (with the selected set of features and the associated solution) is created according to the architecture candidate created previously.

4 Evaluation

This evaluation is to demonstrate the applicability and benefits for several scenarios of (real-world) application environments. With our automated approach, we show how trade-off decisions to select features can be supported automatically with regard to the expected software quality and what effects these decisions may have on the software architecture. For our scenarios, we consider the purpose of *logging*, which is often implemented in practice using the log4j framework. We first model the feature completion corresponding to logging frameworks, including different features and apply the defined structure to two real-world logging solutions, log4j version 1 and log4j version 2. The presented scenarios cover several facets of the design questions that arise from the use of a specific feature in the target architecture such as feature selection, solution selection, feature placement.

4.1 Target System

To demonstrate our approach, we use the model of a community case study, namely the Modular Rice University Bidding System (mRUBiS) [13]. mRUBiS implements a trading and auction platform modelled on the real auction platform ebay.com. mRUBiS has a component-based software architecture and is fully implemented in Enterprise Java Beans 3 (EJB3). The domain model is modelled in the Eclipse Modeling Framework (EMF). As execution engine, mRUBiS uses a GlassFish application server. mRUBiS supports several shops in which goods can be offered for sale. Sellers can offer new items for sale within their shops and check the current inventory. Buyers can register on the platform, log in, search for items using different categories, bid on items, and submit reviews. The mRUBiS model internally consists of nine software components that provide the services. Using `ItemService`, buyers can search for items and place bids. To do this, buyers must first register using the `Authentication` component. The request is then processed using the `Query` component and the `Database` component. Submitted bids are stored in the database using `Persistence`. `UserInfo` lets buyers edit information about their user profile. Sellers use the `Inventory` component to add new items to their shop. This request is forwarded to the database and processed using the query component. `ManageItems` checks the inventory and is then forwarded to the database via `BasicQuery`. The architecture model of mRUBiS has annotations to simulate performance analysis (using (RD)SEFFs) and cost estimation.

4.2 Logger Solutions

A logger collects and records system events, activities, and (inter-)actions over a period and enables tracking and monitoring, statistical analysis or debugging and error recovery. Three feature completion components (FCCs) model the abstract structure of loggers, the `Collector`, `Appender`, and `Formatter`. These components abstract the functionalities and dependencies of the subsystem's software components. In the case of a logger completion, the `Collector` represents the entry point of the logger and receives the log messages. The data are forwarded to the FCC `Appender`, which uses the FCC `Formatter` to convert the logs into a suitable format (e.g. XML) and stores them on a specified write target (e.g. hard disk). Each of the FCCs can have a set of provided and required perimeter interfaces. In the case of the logger completion, the FCC `Collector` comes with several provided perimeter interfaces, while the FCC `Appender` consists of one required perimeter interface. The required perimeter interface of the FCC `Appender` requires an interface to a database if the feature *database logging* is a desired feature. For the evaluation, we model two solutions, namely *log4jv1* and *log4jv2*, and annotate them to the logger completion [8]. These solutions represent variants of the same completion, since they build on each other, but differ in their quality attributes as well as in their realized set of features. *log4jv2* supports a broader range of features and, thus, both versions can be regarded as two different solutions for the logger completion[1]. The two solutions offer both core features, which makes them logging systems. This includes features, such as *FileLogging* and *SQLDatabaseLogging*. However, *log4jv2* offers additional features that we consider as optional. One of these is *AsyncLogging*. For this paper we have concentrated on a subset of all provided features.

4.3 Scenario-Based Evaluation

We studied several scenario-based examples to demonstrate the applicability and benefits of the proposed approach. The scenario covers different design issues in terms of feature alternatives, solution selection, and placement choice. The simulation series considers more than 1000 architecture candidates and evaluates performance and cost of each candidate to find the Pareto-optimal solutions. Here, we evaluate a set of feature alternatives an architect has to consider. The scenario is relevant, but not limited to the requirements engineering phase. Different functional and quality requirements of features must be balanced against each other. Early evaluation of the quality effects of feature alternatives that implement the requirements helps to discuss their

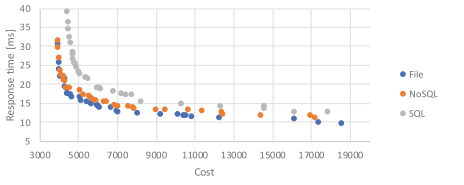

Fig. 1. Comparison of feature alternatives

[1] Please note that the approach is not limited to systems that build on each other and are related in their architecture.

prioritization with stakeholders on a sound data basis. In particular, we compare the features *FileLogging*, *SQLDatabaseLogging*, and *NoSQLDatabaseLogging*. As SQL database, we use MySQL v. 5.7.20 and as NoSQL database, we use MongoDB v. 3.4.10. Both DBMS are configured in the standard configuration. To analyze the scenario, we annotate the components `ItemService`, `Query`, and `BasicQuery` of the *mRUBiS* system with logging. Figure 1 shows the result of the evaluation. The diagram depicts the Pareto-optimal candidates for each feature alternative found by the design space exploration. The candidates with *File* logging show the best quality in terms of response time. The *NoSQL* alternative reaches 7.8% (average) higher response times. The *SQL* alternative is outperformed by the others, namely by 28.4% (average) through NoSQL and by 38.46% through File logging. It should be noted that *NoSQL* and *SQL* logging alternatives also result in slightly increased costs, which is due to the additional database component required by both alternatives. With the results, an architect can decide, which write target of the logger meets the requirements best.

5 Related Work

There are numerous papers that present variability models to define a common architecture for similar solutions. In [3] Atkinson et al. propose their KobrA approach that focuses on component-based product line development. The main component of the KobrA method is a framework that encapsulates a generic description of a family of applications. Here, not only the common parts of an architecture are relevant, but also all differences. They are considered by including all possible characteristics in decision models. These describe options that distinguish between the individual characteristics. If a concrete application is to be developed, the generic framework is instantiated and all decision models are solved. This results in a concrete instance, but does not influence the level of abstraction. There are similar approaches to modeling variability in software (architectures), such as Product Line Software Engineering (PuLSE) [5], the product line design process [6] by Bosch, the FAST [14] approach, or the algebraic language SPLA [2].

6 Conclusion

The approach described here presents a solution for the automatic evaluation and optimization of software architectures in the decision-making process about reusable functionalities. It supports decisions for the selection of features, configuration of features in the software architecture, and different solutions and effects on the quality attributes of the software architecture. The approach is aimed at supporting the software architect in evaluating the effects of features on the quality attributes at development time, even before the actual implementation has been carried out. Through early evaluation, suboptimal decisions can be discarded before implementation, thus supporting more cost-efficient software development. We demonstrated the advantages of this approach using a scenario

from real-world systems. We modeled, analyzed, and optimized different design decisions based on scenarios. The results shown can be used in the next step to implement the software architecture.

References

1. Aleti, A., Bjornander, S., Grunske, L., Meedeniya, I.: ArcheOpterix: an extendable tool for architecture optimization of AADL models. In: MOMPES 2009 (2009)
2. Andres, C., Camacho, C., Llana, L.: A formal framework for software product lines. Inf. Softw. Technol. **55**, 1925–1947 (2013)
3. Atkinson, C., et al.: Component-Based Product Line Engineering with UML. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
4. Bachmann, F., Bass, L., Klein, M., Shelton, C.: Designing software architectures to achieve quality attribute requirements. In: SW Proceedings (2005)
5. Bayer, J., et al.: PuLSE: a methodology to develop software product lines. In: SSR 1999, ACM (1999)
6. Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach. ACM Press/Addison-Wesley Publishing Co., Boston (2000)
7. Busch, A., Schneider, Y., Koziolek, A., et al.: Modelling the structure of reusable solutions for architecture-based quality evaluation. In: CloudSPD 2016. IEEE (2016)
8. Eckert, M.: Konditionale Platzierung von Architekturelementen zur Optimierung von Software-Architekt. Master's thesis, Karlsruhe Institute of Technology (2018)
9. Koziolek, A.: Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes. KIT, Karlsruhe (2013)
10. Koziolek, A., Koziolek, H., et al.: PerOpteryx: automated application of tactics in multi-objective software architecture optimization. In: QoSA-ISARCS 2011 (2011)
11. Martens, A., Koziolek, H., et al.: Automatically improve software models for performance, reliability and cost using genetic algorithms. In: WOSP/SIPEW ICPE 2010 (2010)
12. Reussner, R.H., Becker, S.: Modeling and Simulating Software Architectures: The Palladio Approach. The MIT Press, Cambridge (2016)
13. Vogel, T.: mRUBiS: an exemplar for model-based architectural self-healing and self-optimization. In: SEAMS 2018. ACM (2018)
14. Weiss, D.M., Lai, C.T.R.: Software Product-line Engineering: A Family-Based Software Development Process. Addison-Wesley, Boston (1999)