Reconciling Software Architecture and Source Code in Support of Software Evolution

Thomas Haitzer[a,], Elena Navarro[b], Uwe Zdun[a]

[a] *Software Architecture, University of Vienna, Vienna, Austria*
[b] *Computing Systems Department, University of Castilla-La Mancha, Albacete, Spain*

Abstract

Even in the eighties, the need of managing software evolution has been detected as one of the most complex aspects of the software lifecycle. In this context, software architecture has been highlighted as an integral element of the software evolution process. However, no matter how much effort is put into the architecture, it must eventually be translated into source code. The potential misalignment between architecture and code can lead to failures in the evolution process in terms of economic impacts, failed expectations, and so on. In this article we report on a design science research study that we pursued to answer three research questions. First, we have studied whether and in how far it is possible to design an approach that both enforces the integration between software architecture and source code to avoid architectural erosion and architectural drift and, at the same time, provides automatic guidance to developers to carry out the required change tasks in each evolution steps. Second, we have studied whether this approach may be applied in realistic (open source) cases. Finally, we have analysed whether it is realizable at acceptable costs (in terms of development effort) in comparison to the overall development efforts roughly spent on the evolution of the projects in focus.

Keywords: Components, ADLs, Architecture reconstruction, Evolution styles, M2M transformation, Architectural Knowledge

1. Introduction

Software Evolution is a challenging topic that has attracted a lot of attention during the last decades, since Bersoff et al. [6] and Lehman [41] presented their seminal articles in the area. Even in the eighties, the need for managing the software evolution already has been detected and highlighted as one of the most complex aspects of the software lifecycle. The need for changing software will always exist because of new customer needs, market changes, technology advances, and so on. In this context, the exploitation of software architecture has been highlighted as an integral element of the software evolution process. Several approaches have been proposed that use software architecture as an evolution artifact including different proposals following the Evolution Styles approach [4, 40, 55, 65]. Authors claim that the evolution from an initial to a target architecture should be carried out by planning and analyzing different evolution paths, so that the risks and problems during the evolution can be avoided or at least mitigated.

Despite the undoubted advantages that such an approach has, without tight integration with the software development activities, it is unlikely to be put into practice. As Ozkaya et al. [56] found out during

Email addresses: thomas.haitzer@univie.ac.at (Thomas Haitzer), elena.navarro@uclm.es (Elena Navarro), uwe.zdun@univie.ac.at (Uwe Zdun)

URL: http://informatik.univie.ac.at/thomas.haitzer (Thomas Haitzer), +43-1-4277-78521 (Thomas Haitzer), +43-1-4277-8-78521 (Thomas Haitzer), https://www.dsi.uclm.es/personal/ElenaNavarro (Elena Navarro), +34-967599200-2365 (Elena Navarro), http://informatik.univie.ac.at/uwe.zdun (Uwe Zdun), +43-1-4277-78510 (Uwe Zdun), +43-1-4277-8-78510 (Uwe Zdun)

their interview study, developers in practice do not use architecture-centered practices to manage evolution decisions. They just focus on the coding efforts when evolution needs arise, so that finally architects must cope with architectural erosion and architectural drift [59]. Better support for developers for integrating their work with the software architecture is needed to avoid architectural erosion and architectural drift. This is also pointed out in a recent study of 705 official releases of nine open-source software projects by Neamtiu et al. [54].

The fact that developers only pay attention to code is especially common in Open Source projects where the attention on planning or modeling is often even non-existent. An example in this sense is Moodle, one of the most widely used platforms for e-learning. It has a wide community of developers, as well as a properly described development process. Moreover, it also has had major changes along its evolution process [46] as well as large numbers of issues filed for specific versions [72]. However, there is not a clear description of the architecture, the planned evolution, its traces to code, and so on. That is, if we have a look for example on Moodle's roadmap for the latest Version 2.7 [48], we find that it is described just textually in terms of new components to be developed (e.g. a new events system or a new framework for logging and reporting) or components to be changed (e.g. changes to bootstrap themes). This lack of architectural guidance can lead to a misalignment between the architecture as planned by (some of) its developers and the concrete architecture (implementation) really coded by the developers. But even when the architecture is clearly described this misalignment between code and software architecture frequently occurs. For instance, Nakagawa et al. [51] present a case study that shows clearly that the severe differences between the conceptual architecture of a system and its concrete architecture as implemented were accidentally or inevitably inserted as the code evolved. Further, the authors claim, these differences had a very negative impact on the system evolution affecting important quality attributes, such as functionality, maintainability, and usability.

The previous issues led us to the observation that a combination of measures addressing architectural erosion and architectural drift with clear architecture-guided evolution planning for the source code seems to be needed – two aspects that were so far treated only separately in the literature. Hence, our first research question addresses the feasibility of such an approach:

RQ1: *Is it possible to effectively combine enforcing the integration of software architecture and source code to avoid architectural erosion and architectural drift with architecture-guided evolution planning? If yes, how can it be achieved?*

As from the study of the state-of-the-art literature it is not possible to claim whether a tight combination of the two goals is possible at all, this first research question concerns the feasibility of the approach (i.e., can it be done at all?). Consequently we selected a research method focused on the design of novel or innovative artifacts: design science research. Design science research [27, 75] is an established research method in which first a research question is posed, and then an develop/evaluate cycle is continuously repeated until a satisfactory solution for the research question has been obtained. In the course of this research, the research question can be altered or refined. In the first iterations, usually simplifying assumptions are made, which are stepwise removed during later iterations.

For the evaluation of the progress in the research study iterations, we have applied our research in four case studies of non-trivial evolutions of Apache CXF and Soomla. We have used the experiences from those cases to continuously improve our approach and also used them to show the applicability of our approach in real-life software system evolutions. This evaluation was conducted to answer our second research question, that concerns the applicability of the approach in larger examples than toy examples:

RQ2: *If RQ1 can be positively answered, is it possible to apply the approach in realistic software projects?*

In addition, we have quantitatively analyzed the effort required to apply our approach in these four cases. This is required and interesting to study, because even if RQ1 and RQ2 can be positively answered, we still need to assess and justify the costs of our approach in order to make it applicable in real-life projects. That is, it is unlikely that our approach will ever be applied in real-life projects, if it requires a substantial extra effort during development. Here, we argue that the effort required for our approach must be compared to the development efforts spent on the evolution of the system in focus itself. This led us to our third research question:

RQ3: *If RQ1 and RQ2 can be positively answered, is it possible to apply the approach with little extra effort compared to the effort roughly spent on the evolution of the software system in focus?*

This article claims that a positive answer to RQ1 can be provided by integrating three different approaches, namely Evolution Styles, Architectural Knowledge and Architectural Reconstruction. In particular, we suggest using evolution styles to guide the stepwise architecture evolution in a number of incremental evolution steps. In each evolution step, we suggest using an architecture reconstruction tool to enforce the evolution of architecture and source code, keeping both in sync. The reconstruction tool uses an architectural abstraction specification to generate a component view from the implemented source code. We suggest realizing each evolution step by (a) transforming the architectural abstraction specification for the architectural reconstruction to contain the architectural changes and (b) developing the source code accordingly. After both, the evolution of the architecture and the code have happened, we automatically reconstruct the architecture using our reconstruction tool. The tool now can detect inconsistencies between the architecture specification and the source code. Such inconsistencies are the result of violations in the architecture, the transformation or the source code. These artifacts are incrementally refined until no more violations occur. The result is that the evolution step is carried out and that the architecture and the source code are in sync.

With regard to RQ2, we show the applicability of our approach in four case studies of non-trivial evolutions of Apache CXF and Soomla. Regarding RQ3, we further show for those cases that the required efforts to apply our approach are minimal compared to the development efforts roughly spent on the evolution of those systems. Although these quantitative results provide only a rough estimation based just on the effort needed for the evolution of Apache CXF and Soomla, the order of magnitude to which our approach requires less effort than such roughly estimated efforts indicates that our results can be likely generalized at least to some extent.

This work is structured as follows. After this introduction, Section 2 describes the main foundations of this work, and Section 3 analyses the related work. Section 4 provides an overview of our approach, and Section 5 describes the technical details of the tool support of our approach. Then four case studies are described in Section 6 to illustrate the applicability of our approach. Finally, we discuss our results in Section 7, and the main conclusions drawn as well as our future work are presented in Section 8.

2. Background: Architectural Styles and Architectural Knowledge

Since the latest definition of the Lehman's Laws of Software Evolution [42], many empirical studies on their validation have been published. All these studies, as Herraiz et al. summarize [25], validate the first law of evolution, that is, software "must be continually adapted, or else it becomes progressively less satisfactory in use." Mainly, we can consider that systems must evolve in order to continue being useful. This confirms the need of techniques, methods, tools, etc. guiding the different stakeholders in the software evolution process. There are different approaches that facilitate such software evolution. Several studies [2, 4, 10, 28, 55] claim that architecting is the most appropriate point of view to address this issue as it enables stakeholders to work at a higher abstraction level.

There is a great deal of attention on architecting for software evolvability, especially since 2008. Several authors [10, 31] have pointed out that this can be due to the fact that more and more systems turn into legacy systems, so that both practitioners and researchers have realized the importance of software architecture evolution. This attention on software architecture evolution is also shown by the number of literature reviews and mapping studies that have been published during the last five years. Some of them [10, 31] have analyzed the available architecting practices and have classified them mainly into two different groups: (i) those practices that help stakeholders to design [68] and assess [12] the evolvability of the software architecture when this is being specified; (ii) those practices (see e.g. [4, 40]) applied to evolve the software architecture. Our approach focuses on the latter category, that is, on providing architects and developers with assistance when both software architecture and code must be evolved.

Different approaches have been proposed that pursue to facilitate this evolvability. One of them has been the definition of *Evolution Styles*. Basically, an Evolution Style defines the constraints to be fulfilled by any possible evolution of a system. They resemble Architectural Styles as defined by Perry and Wolf

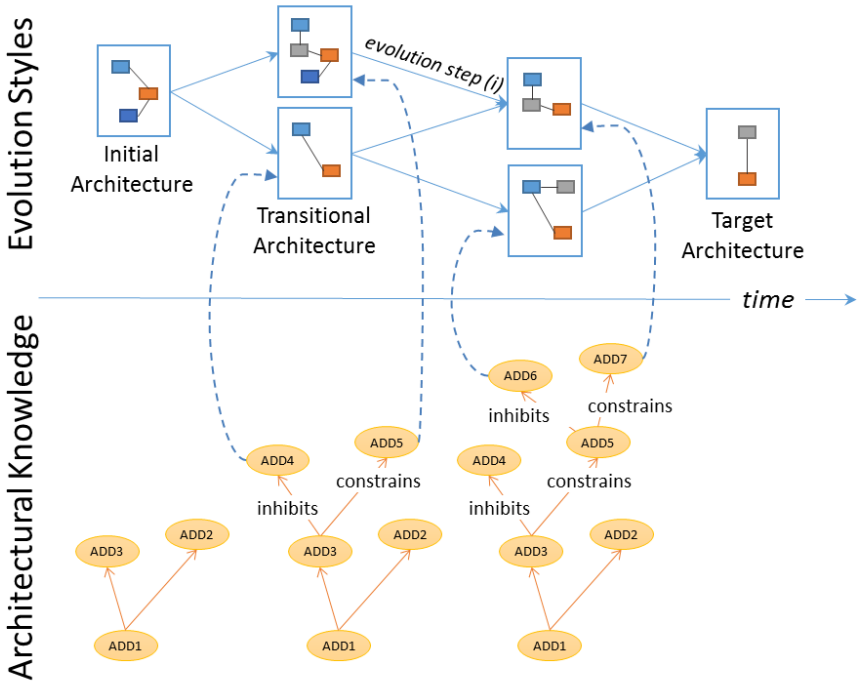

Figure 1: Evolution Styles and AK guiding the evolution: Architectural Design Decisions (e.g. ADD1), snapshots of the architecture (e.g. initial architecture, transitional architecture), evolution steps (e.g. arrow from initial architecture to transitional architecture), and traceability links (e.g. dashed line from ADD7 to evolution step (i))

[59], in the sense that they are also prescriptive rather than descriptive. Therefore, while an architectural style prescribes the architecture, an evolution style prescribes its evolution. These styles are defined when a need to evolve from an initial architecture to a target architecture emerges, as shown on top of Figure 1. In order to define an Evolution Style, architects identify different *evolution paths*, that is, different alternatives to facilitate such evolution. Each evolution path is defined as a sequence of *evolution steps*. As Figure 1 illustrates, every evolution step is specified by means of an *evolution operator* that determines an atomic modification of the architecture. As a result of every evolution step a transitional architecture is obtained, that is, a new snapshot of the software architecture. As can be noticed, every step imposes some new constraints on the architecture, so that the evolution of the system must satisfy each one of the constraints imposed by such evolution steps. For instance, as described in Case Study 2 (see section 6.2), for evolving the architecture from Soomla v3.2 to Soomla v3.3, three different evolution steps were identified: (1) Replacing Google-independent BillingProvider with Google-specific Billing component; (2) Adding the new GooglePlay-Billing component; (3) Adding the new AmazonBilling component. To carry out each one of these evolution steps, different evolution operators are applied that automate them and support the prescriptive nature of the evolution style.

Regarding the evolution of the software architecture, a second important aspect to be considered is how it became specified in its latest form. It is important to know not only which or how many components and connectors it has, or how they are related among themselves, but also it is important to know what decisions have been made and why, that is, the rationale [59] behind the architecture specification. This is usually referred to as Architectural Knowledge (AK) today. As Figure 1 illustrates, while applying an Evolution Style, each evolution step corresponds to a specific architectural design decision (ADD). According to the ISO 42010 standard an architectural decision affects one or more architectural elements and pertains to one or more concerns [30], which is often documented in the form of ADD templates. Therefore, an ADD conveys the decision behind an evolution step thanks to the necessary traceability link among them (see

dashed line from ADD 7 to evolution step (i) in Figure 1). As will be explained in Case Study 2, (see section 6.2), three different ADDs (Figure 5, Figure 10 and Figure 12) were defined that convey to both architects and developers how and why Soomla was evolved from v3.2 to Soomla v3.3. An important aspect of these ADDs is that they are used as coding guidelines in our approach so that they can guide developers during the change task to be carried out. Moreover, as can be observed in Figure 1, not only these ADDs but also the relationships (inhibits and constraints) among them must be recorded. For instance, as shown in Figure 1, ADD5 has an *inhibits* relation with ADD6 and a *constrains* relation with ADD7, that is, this determines that the evolution path selected will include the evolution step (i). Moreover, at the bottom of Figure 1 it is shown that the AK is evolving along the time from three ADDs to seven ADDs that describe how the evolution style is applied. Therefore, all the history of the evolution of the system is properly described, so that the AK is used both to guide the evolution and to avoid the previously sketched problems in the evolution process.

3. Related Work

As aforementioned in the introduction, this work has not been defined from scratch but from the integration of three different approaches: Evolution Styles, Architectural Knowledge (AK) and Architectural Reconstruction. Each of them is described in the following sections highlighting the advantages they provide as well as the shortcomings that prevent them from being used in isolation as an answer for the software evolution problems explained in Section 1. Moreover, those works directly related to our proposal are also analyzed considering their advantages and disadvantages for evolution. Finally, a comparison between the different approaches and ours is presented in Section 3.4.

3.1. Techniques for Evolving Architectures

The above-mentioned literature reviews [10, 31] identify a wide spectrum of approaches for evolving architectures. Among them we can find approaches [29, 14] that have analyzed the exploitation of transformation techniques for evolving legacy systems. Specifically, they transform the architecture, previously abstracted from the code, to apply all the necessary changes and then transform it again to code. Therefore, all the architectural evolution is carried out just in a single step, without keeping the knowledge behind each evolution decision, analyzing the different alternatives during the evolution, etc. Such transformations from architectural specifications to code are not always applicable, especially when the evolution to be carried out entails the development of new code.

A number of studies [40, 55, 65] have used the concept of architectural styles as the way to constrain the changes that can be applied to a system being evolved. They define these architectural styles as sets of evolution patterns, which are specified as generic transformations. Despite the importance of AK during the evolution process, highlighted by the authors [40, 55], no support or assistance is provided for its use. A similar proposal has been presented by Barnes et al. [4, 5]. They define Evolution Styles to support software architects while evolving a software system from an initial architecture to a target architecture. As shown on top of Figure 1 (see Section 2), while applying the approach, several evolution paths can be defined as sequences of evolution steps. Every evolution step is specified by means of an *evolution operator* that determines how the architecture must be modified. One of the most interesting aspects of this approach is that it provides two different kinds of analysis: path evaluation functions and evolution path constraints. In order to carry out the former analysis, every evolution path is annotated with a list of properties, such as duration or cost, so that the software architects can determine, for instance, the economical or temporal feasibility of an evolution path. On the other hand, and thanks to the automatic support for the analysis of evolution path constraints [5], the software architects can determine whether the identified evolution paths satisfy the rules of the evolution style. For instance, they could check that a component is not removed until a wrapper has been developed. One of the main differences regarding our approach is that their evolution operators have not been automated so that they cannot be reused in different systems. However, in our proposal every evolution operators is implemented by means of a QVT-o transformation (see Appendix A), that is, an operation to modify the architecture specification, such as delete a component, add a connector, etc.

As can be observed, the use of evolution styles helps software architects to plan the evolution of the software architecture that should be subsequently implemented by the developers. Moreover, they can also evolve a system in a way that does not violate any of the rules described in the style. However, so far the relation to code that implements the architecture has been neglected and in particular that the code might not be compliant with the planned software architecture. Ensuring the consistency between architecture and code after evolution steps is another novel contribution of our work. Specifically, we focus on extending the evolution steps to facilitate the evolution of software architecture and code, keeping both in sync.

3.2. Architectural Knowledge

During the last decade, a great deal of attention has been paid to the field of Architectural Knowledge (AK), since Bosch [8] recalled the attention to this important aspect. Unfortunately, too frequently, this AK is vaporized as architects fail to describe it in a proper way. As Harrison et al. state [24], this vaporization can have critical consequences, the expensive system evolution being one of them.

Several empirical studies support this argument. For instance, Bratthall et al. [9] carried out a survey with 17 subjects from both industry and academia and concluded that most of the interviewed architects considered that by using AK they could shorten the time required to perform change tasks. The interviewed subjects also concluded that the quality of the results, when they had to predict changes on unknown real-time systems, was better using AK. Ozkaya et al. [57] have also concluded, during their interview study, that the difficulties during both the initial phases and the evolution of systems are not only due to the unavailability of AK, but also depend on its ineffective use. This is especially noteworthy because, despite this unavailability and ineffective use of AK, the interviewed architects have also remarked that they perceive AK to be essential when evolution hits. Feilkas et al. [19] have carried out a case study on three industrial information systems and considered the importance of AK as one of their research questions. The authors have detected that one of the problems in these projects was that developers were not aware of the intended architecture because the AK was not properly described.

On the other hand, other empirical studies [1, 17, 43, 66, 71] have focused their attention on analyzing available AK practices. As one of their main results all of them emphasize its usefulness as a valuable artifact for the software architecture evolution. For instance, some works [16, 58] have exploited an ontology-based approach to describe AK that can be used for impact analysis by means of if-then scenarios. Another approach called Architecture Rationale and Element Linkage (AREL) [67] is basically a language used to describe causal relationships between Architectural Design Decisions (ADDs) and architectural elements. Using these descriptions, the approach exploits Bayesian Belief Networks to carry out three different probability-based reasoning methods that enable architects to estimate the impact of a change before it is carried out. Other interesting work is a family of languages called ACL [70] that enable software architects to relate architectural choices to decisions that are made throughout the development process. One of the main strengths of the approach is that these decisions are defined as constraints, so that the architectural model can be checked after its evolution to determine its conformance. Another approach [53] proposes the use of anti-patterns to detect and record conflicting ADDs while the architecture is being evolved. However, none of these approaches really considers the evolution of the architecture, but just the use of AK for analysis purposes.

Recently Cuesta et al. [15] have presented a proposal called AK-driven evolution styles (AKdES), which focuses on the bottom part of Figure 1. Basically, the authors claim that every evolution step is carried out because an evolution condition has been triggered. Then an evolution decision, that must be stored as part of the AK as an ADD, has to be made as reaction to the evolution. This evolution decision leads to an evolution step that must be carried out. As a result, ADDs are used to convey the ideas behind each evolution step to both software architects and developers. However, the already stated misalignment between code and software architecture can also happen, despite considering such AK as part of the evolution process, because misunderstandings, as well as architectural erosion [69] and architectural drift [61], can, accidentally or not, occur. The approach presented in this article aims to address this open issue.

3.3. Architectural Reconstruction

The interest on the evolution of code and architecture has increased during the last decade, especially in the field of software architecture reconstruction. A significant number of architecture reconstruction approaches focus on the automatic recovery of the architecture from code by identifying architectural patterns and design patterns [26, 39]. Other approaches focus on identifying components or similar abstractions through automatic clustering [74] like the one proposed by Corazza et al. [13] which is based on latent semantic indexing on text found in source code. A variety of approaches establish different kinds of abstractions between source code and the architecture level. Some use graph-based techniques [62] while others utilize model-driven techniques [64, 49, 18, 37], or logic oriented programming [47]. Other approaches like the one by Ganesan et al. [20] analyze external dependencies to discover architecture and analyze a system's quality attributes.

Knodel et al. [38] conducted an experiment on live compliance checking, where the teams using the live compliance checking violated the planned architecture significantly less then the ones without. Lindvall et al. [44] report their experiences with the successful induction and usage of the SAVE architecture evaluation framework at the Johns Hopkins University Applied Physics Laboratory.

All the presented approaches for software architecture reconstruction, as far as we know, consider the evolution of software architecture and code just in a single step. That is, they do not consider evolution as a process in which software architects and developers must cooperate. They just focus on creating or identifying new abstractions from code, without considering how they should be exploited in the context of an evolution process.

In our previous work [22] we proposed an approach for creating an architectural component view based on object-oriented source code using a domain specific language (DSL) that we call Architecture Abstraction DSL. This approach focused on supporting the software architect throughout the evolution of a software system. For this task we realized consistency checking between the architectural component view defined in the DSL and the underlying source code as well as the automatic generation of traceability links between the architectural elements and the source code artifacts. Our approach is independent from the source programming language (by working with an intermediate representation of the source code in form of an automatically reconstructed UML class model) and aims at providing a flexible yet simple way to define the relations between architectural elements and source code artifacts.

While we were able to show the applicability of that approach for a number of case studies that analyzed the changes necessary to update the definition of the architectural component view between different versions of existing open source software, the approach did not take the rich architectural knowledge captured in ADDs into account and therefore might not be successful in preventing the evaporation of architectural knowledge. In this article, we propose to utilize this DSL in a top down fashion, where the software architect uses a variant of the Architecture Abstraction DSL to define an architectural component view which allows the definition of coding guidelines for each of the defined architectural elements. We then use these guidelines to check whether the source code reflects the intended architecture using traceability links and consistency checks similar to those in our previous work.

3.4. Summary of Contributions in Comparison to the Related Works

Table 1 summarizes the main differences of the contributions of our approach compared to the works discussed in the previous sections. In our literature study, we have identified five main features that have driven this comparison:

- *Architectural transformation.* Only three related approaches exploit like our approach architectural transformations to automate the process. The model-driven approach, followed by two related works [29, 14], is the only other approach that intends to generate code. The other approaches use the transformations as a way to automate the evolution steps to be carried out (which is also a goal of our approach).

- *Evolution planning.* Only three related approaches have been defined to assist architects in the process of planning the evolution, which is one of the main contributions of our approach.

- *Analysis.* Different approaches provide some kind of analysis assistance. While there are other approaches [47, 50] that check consistency between architecture and source code, our approach is the only one that provides support for checking whether there is a misalignment between code and software architecture, in order to avoid architectural drift and erosion that specifically considers evolution.
- *AK in the evolution.* Several approaches exploit architectural knowledge for evaluation purposes, but they do not drive the evolution process as a result of its exploitation. Only AKdES and our approach focus on this exploitation of the architectural knowledge.
- *Coding guidelines.* Our approach is the only one that assists architects in conveying coding guidelines to developers. This feature and the support for checking are considered critical to avoid the likely misalignment between the planned architecture and the concrete architecture really coded by the developers.

In summary, our approach makes a number of unique contributions in comparison to the related work. It is also unique that it is the only one approach that provides support in all five identified categories.

Approach	**Architectural transformation**	**Evolution planning**	**Analysis**	**AK in the evolution**	**Coding guidelines**
Model-driven approach [29, 14]	From architecture to code	-	-	-	-
ETAK [40, 55, 65]	Architecture evolution	Supported	-	-	-
Automated planning [4, 5]	Architecture evolution (not automated)	Supported	Alternative evolution steps	Architecture evaluation	-
Ontology-driven [16, 58]	-	-	Impact analysis	Architecture evaluation	-
AREL [67]	-	-	Impact analysis	Architecture evaluation	-
ACL [70]	-	-	Architectural checking	Architecture evaluation	-
Anti-patterns [53]	-	-	Detect conflicting decisions	Architecture evaluation	-
AKdES [15]	-	Supported	-	AK as evolution driver	-
Model-based architectural reconstruction [18]	-	-	Model-based consistency checking	-	-
Static architecture evaluation [37]	-	Conclusions from evaluation	Static evluation	-	-
Automatic clustering [74, 13]	-	-	Identify clusters in source code	-	-
Software Reflexion Models [49]	-	-	Detect architectural drift	-	-
Pattern identification [26, 39, 62]	-	-	Detect pattern instances	-	-
Source code views [47]	-	-	Violated predicates	-	Guidelines through logic predicates
Our approach	Architecture evolution	Supported	Architectural and code checking	AK as evolution driver	Supported

Table 1: Comparison among different proposals

4. Our Approach: Code and Software Architecture Evolution

4.1. Research Method

The main goal (RQ1) of our research study is to investigate whether and in how far it is possible to design an approach that supports enforcing the integration between software architecture and source code, to avoid architectural erosion and architectural drift, and at the same time provides automatic guidance to developers to carry out the required change tasks in each evolution step. To address this research question, as its main focus is on the design of novel or innovative artifacts, we have carried out a design science research study [27, 75]. Design science research produces different outputs: constructs, models, methods, and instantiations [73]. While constructs are the conceptual vocabulary of a domain, models are a set of propositions expressing relationships among constructs. Methods are a set of steps to perform a task, and instantiations are operationalized models, constructs, and methods. Design science research is comprised of five steps [73]:

1. **Awareness of problem**: This might result from different sources like e.g. earlier research efforts or other disciplines and it results in a proposal. We have identified the literature sources summarized in Section 3 in this step to understand the field, and specifically observed that no solution exists that effectively combines enforcing the integration of software architecture and source code with architecture-guided evolution planning for the source code in each evolution step.
2. **Suggestion**: This is "a creative step where new functionality is envisioned based on a novel configuration of either existing or new and existing elements" [73]. This step results in a tentative design. We designed the approach described in the next section, which combines three existing previous approaches: Evolution Styles, Architectural Knowledge (AK) and Architectural Reconstruction.
3. **Development**: Here, the Tentative Design is further developed and implemented. We realized a prototype tool to be able to test our approach.
4. **Evaluation**: The constructed artifact is evaluated according to criteria that are implicit (or sometimes explicitly) mentioned in the proposal. In our case, the applicability in four realistic cases (RQ2) as well as the cost effectiveness of the approach in terms of development effort (RQ3) were the main guiding evaluation targets. The evaluation employs case study research [75] as a research method, with quantitative estimations of development efforts for the four cases.
5. **Conclusion**: In this step, the evaluation results are judged to be sufficient or insufficient. They are also consolidated and written up. This last step might create additional iterations in the research loop. We were able to conclude that we indeed can suggest an approach to satisfy RQ1, apply it in the four selected open source cases (RQ2) with minimal effort compared to the overall development effort necessary in the projects (RQ3).

We have performed the main steps in 5 iterations, first an initial iteration with toy examples, and then one additional iteration per selected case.

4.2. Approach Overview

As described in Section 1, our approach relies on three previous proposals: Evolution Styles, Architectural Knowledge (AK) and Architectural Reconstruction. The approach uses Evolution Styles [4] to help the architect plan and execute the evolution from an initial architecture to a target architecture following one of the different evolution paths available. It also takes into account AKdES [15] so that each evolution step is traced from a specific ADD that describes how it must be carried out, keeping the history of the evolution process. However, it differs from both of them in how these evolution steps are carried out. Specifically, it extends the definition of both proposals to provide software architects and developers with support for the evolution of both artifacts.

As Figure 2 illustrates, in our approach, an evolution step does not focus only on the architectural transformation or the architectural decision made but also on the code as well. In particular, our approach considers an evolution step as an iterative process that entails several tasks, artifacts and tools. The assumption of our approach is that the component view of the architecture is not manually created, but

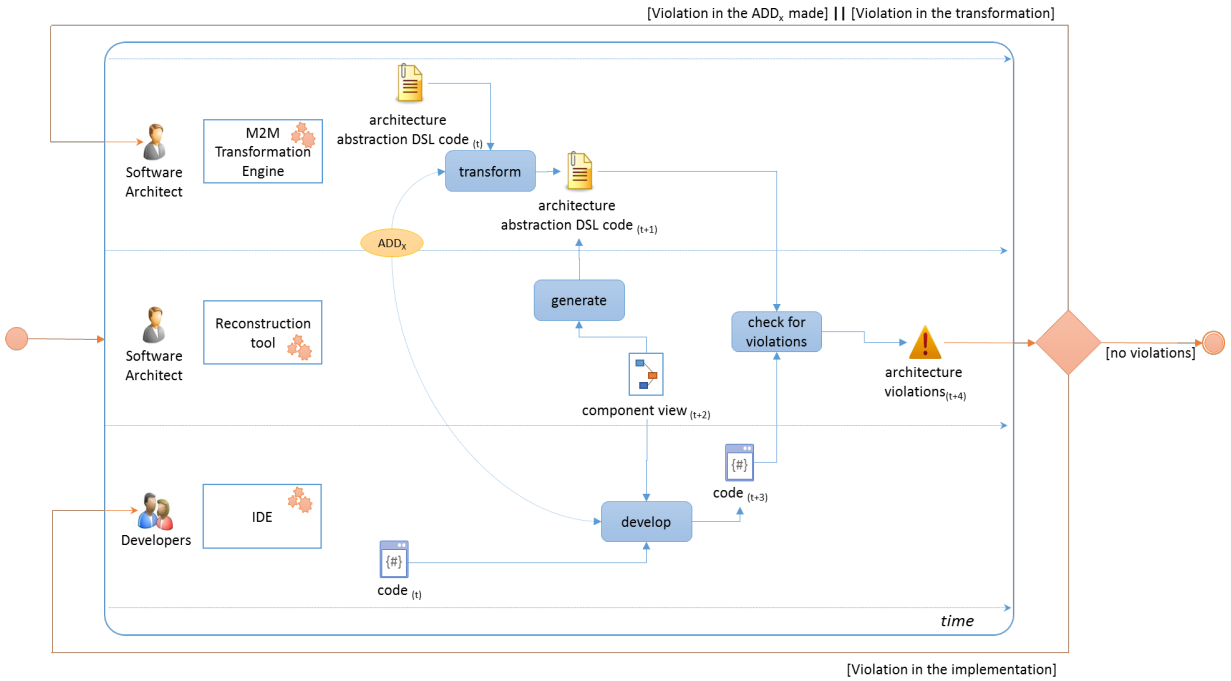

Figure 2: Roles (e.g. software architect), tooling (e.g. M2M transformation engine), artifacts (e.g. architecture abstraction DSL code) and activities (e.g. transform) to carry out an *evolution step*

that the Architecture Reconstruction Tool (see Section 5) is used to automatically create the component view based on the architecture specification made in the Architecture Abstraction DSL and on the source code.

Assuming that we are in the beginning of an evolution step (in instant t in Figure 2) we need the following artifacts: A description of the architecture using Architecture Abstraction DSL, an Architecture Design Decision (ADDx) to be applied in the next evolution step, and the snapshot of the code (that is consistent to the current architectural description). While the description of the architecture usually exists from the last evolution (when using our approach), the architect needs to create it using our Architecture Abstraction DSL if our approach is applied for the first time. If all these artifacts exist the following tasks should be carried out:

1. First, the software architect evolves the architecture into a new version in order to apply the ADDx made. Following the ideas presented in [3, 45, 52], this evolution is carried out by applying Model-To-Model (M2M) transformations, that is, the architect manually maps every ADD to a set of defined architectural transformations. This means that the evolution graph is really described as a graph of architectural transformations, where each architectural transformation applies the changes of one ADD to the architectural description. This helps to improve the maintainability of the evolution graph because it can be re-generated/updated in an automatic way whenever it is necessary.
2. Once the architecture has been evolved, the software architect generates, using the Reconstruction Tool [23], a UML component view from the software architecture specification. Developers are provided with this component view because, according to our previous empirical studies [22], it helps to improve the design understandability.
3. Then, the developers develop a new version of the code using both the component view as well as the ADDx that was defined by the software architecture beforehand. These artifacts also describe how the evolution step should be carried out, from the source code point of view. In other words, the specification in the transformed Architecture Abstraction DSL can be seen as coding guidelines for the changes that need to be made to the source code in order to realize ADDx.

4. Once, the code evolution has been finished, the software architect uses the Reconstruction Tool to check whether any misalignment between the code and the planned architecture exists.

The software architect analyzes the results of the Reconstruction Tool and determines whether any one of the following violations has happened:

- A violation in the implementation. Developers failed to apply properly the required ADDx or the coding guidelines provided in the Architecture Abstraction DSL. Thanks to the facilities provided by the Reconstruction Tool, the software architect knows which areas of the code fail in the implementation step, and provide developers with the necessary warnings. Then, a new iteration would be carried out by the developers in order to solve the detected problems in the implementation.

- A violation in the ADDx was made. When the software architect evaluates the violation, he can also conclude that it was a problem in the decision made as it cannot be properly traced to the code. Therefore, a new decision must be made (ADDy) that guides the developers properly and considers the knowledge gained during the implementation of ADDx. The previous decision ADDx should be marked as inhibited, establishing the rationale behind that decision. A new iteration of the evolution step would then be carried out by software architects and developers to apply the new decision made.

- A violation in the architecture abstraction specification. Finally, the software architect can conclude that the architecture abstraction specification created as result of applying the ADDx was not appropriate and a correction of the coding guidelines provided through the Architecture Abstraction DSL is needed in order to properly align ADDx and the corresponding source code.

As a result of the checking task, the software architect can be also informed that no violations were detected. This means that the evolution step was successful and a new version of both the source code and a corresponding Architecture Abstraction DSL specification is available for realizing the next evolution step of the evolution path.

As can be observed, both software architects and developers can reconcile the different views they have of the software evolution, software architecture and code, respectively. Thanks to the introduction of the Evolution Styles, software architects can plan and analyze the evolution of the software architecture. Then, by using the AK they can convey the idea behind each evolution step to the developers and also keep the history of all the decisions they made. Finally, thanks to the Architectural Reconstruction approach, both software architects and developers become aware about whether any kind of problems exists during the on-going evolution step. As a result of the integration of the three approaches, Evolution Styles, AK and Architectural Reconstruction, both software architects and developers are closely cooperating in the process of evolving code and software architecture.

To apply the proposed approach, a small number of tools are suggested and/or required in addition to those of the usual software development process. One of them is the Reconstruction tool, which enables one to identify the description of the architecture and checking the source code against the defined architecture. While we use our own prototype it might be possible to use other architecture conformance checking techniques to replace our Reconstruction tool. In addition, while it could be done manually, we strongly suggest a model-to-model transformation for updating the architecture description of the system. This ensures that all changes are documented so that they can be easily retraced. To document the ADDs that describe the rationale of the changes, we also propose to use a tool. For our setup, we have used the new version of AdViSe as we are already familiar with this tool, but any other tool for documenting architecture design decisions can be used. Last but not least, a tool for writing the source code is necessary as in every software development project. We suggest to use a single tool-suite that integrates as much of the tools a possible. For our setup, all tools are based on Eclipse so that an easy integration of the different tools was possible.

In the following section, the approach details and the tool support of the approach are described. Next, in Section 5, several case studies are used to present how the approach was put into practice.

```
Component Demo
        consists of
        { // brackets for overriding operator precedence
        /* Structure based: include
         * everything inside this package*/
              Package(org.example)
              and not /* compositon - set difference*/{
                    /* Name-based: classes with
                     * name containing No*/
                    Class(".*No.*")
              } or /* composition - union */ {

        ChildOf(org.example2.AbstractSuperTypeClass)
              }
        } and/* composition - intersection */
        InstanceOf(org.example.IExampleInterface)
```

Figure 3: Example of component of the Architecture Abstraction DSL that contains at least one rule of each category.

5. Approach Details

In this section we describe the technical aspects of our approach and especially the tool support for our approach in more detail. Our approach assumes that ADDs have been made and documented before an evolution step happens. Any ADD documentation tool can be used for this task. In our work, we use the ADvISE[1] tool for this task. As already described above, we created an Architecture Abstraction DSL in our previous work that focused on Software Architecture Reconstruction. We now utilize this DSL to describe the architecture specification. The Architecture Abstraction DSL is written in Xtext[2] and allows specifying architectural components and connectors and their relations to source code using a number of different rules. These rules can be grouped into four different categories:

- Rules working on existing structures in the source code: The Architecture Abstraction DSL supports to relate an architectural component to specific packages, classes, and interfaces that exist in the source code. One example for this category is the Package rule shown in the example in Figure 3 which selects everything inside a specific package.
- Rules utilizing existing relations between source code artifacts: Here the Architecture Abstraction DSL explicitly supports sub- and supertype relations for classes and interfaces, interface realizations, and other dependencies. Figure 3 shows the ChildOf rule as an example for this category. This rule matches all classes that extend the referenced class.
- Rules that operate on the names of source code artifacts: Regular expressions on the names of the source code artifacts can be used to relate an architectural component to all packages, classes, or interfaces for which a given expressions evaluates to true. One example for this is the rule Class(".*No.*"), a rule for matching a regular expression over all class names (as shown in Figure 3).
- Composition of rules: In addition, more complex definitions are supported through the implementation of the three set operations, that is, union (or), intersect (and), and difference (and not), which are all used in Figure 3.

During the evolution of a system, after an architecture has been specified, every ADD that is made, requires us to change the architecture specification. We specify changes to the architecture abstraction specification in terms of QVT-o (Query/View/Transformation-operational) transformations[3]. QVT-o enables us

[1] https://swa.univie.ac.at/Architectural_Design_Decision_Support_Framework_%28ADvISE%29
[2] https://www.eclipse.org/Xtext
[3] https://www.eclipse.org/mmt/?project=qvto

```
modeltype DSL uses 'http://www.univie.ac.at/cs/swa/component/
architectureabstraction/ArchitectureAbstractionDSL';

transformation addComponentTransformation
      (in componentview:DSL, in newComp:DSL , out output:DSL);
main() {
componentview.rootObjects()[DSL::Transformation]-> map
      addComponent(newComp.rootObjects()[DSL::Transformation]
      ->asOrderedSet()->first().map getComponent());
}

mapping DSL::Transformation::addComponent(in inputComp:DSL::ComponentDef):
DSL::Transformation {
      result.name:=self.name;
      result.components:=self.components->including(inputComp);
}
mapping DSL::Transformation::getComponent () : ComponentDef {
init  {
      result := self.components->asOrderedSet()->first();
      }
}
```

Figure 4: QVT-o transformation for adding a component to the architecture specification

to define transformations for any form of EMF Models. As our architecture specification is implemented as an Xtext DSL, no setup for QVT-o was required. We implemented QVT-o transformations for the following changes[4] to the architecture specification:

- addComponent: Creation of a new architectural component. The implementation is shown in Section 6.3 in Figure 4.
- deleteComponent: Removal of an architectural component. Its source code is shown in Appendix A in Figure A.22.
- addConnector: Creation of a new connector between two architectural components. The implementation is shown in Appendix A in Figure A.23.
- deleteConnector: Removal of a connector between two architectural components. Figure A.24 and Figure A.25 in Appendix A show the QVT-o code for this transformation.
- updateAbstractionSpecification: This transformation is used to replace the existing architecture abstraction specification of a component with a new specification. The transformation's implementation is shown in Appendix A in Figure A.21.

As an example for these transformations (see Appendix A for their specification) we show the transformation for adding a new component to the architecture specification in Figure 4. The addComponent transformation takes two input models: one is the existing architecture specification and the second model contains the component to be added. This transformation then generates the modified architecture specification as an output model. These transformation can be used by the architect to make all the necessary changes to the architecture, and ensure that each evolution step is retraceable.

Once the architecture specification has been changed, the Reconstruction Tool, which is built around our Architecture Abstraction DSL, transforms our architecture specification into a UML component view (using

[4]As the Architecture Abstraction DSL currently does not support the definition of ports, no transformations for creating or deleting ports in the architecture specification have been defined at the moment.

Xtend[5] for generating the model). In addition the Reconstruction Tool performs a number of consistency checks on the architecture specification and source code and any resulting issues are listed in a consistency report. This report is implemented as warnings and errors in the user interface for our Architecture Abstraction DSL. For example, among many other verifications, it checks for connectors between components that exist in the source code but do not exist in the architecture specification and parts of the architecture specification that do not relate to any source code artifacts. Whenever the source code changes, the checking can automatically be executed and a new consistency report is generated. This supports the software developer during and the software architect after the implementation step of our approach when they need to confirm whether the evolution of architecture and code was successful or whether inconsistencies between architecture specification and code still exist. An example of a consistency report is shown below in Section 6.3, Figure 16.

6. Case Studies

As described in Section 4.1, our approach has been designed following the design science research guidelines, and an *evaluation* is one of its steps. For this reason, four case studies have been carried out. Although all of them pursue to answer the research questions, that is, the evaluation of the approach using real systems, each one was applied to perform such evaluation under different conditions. In first two case studies, we applied our approach in the architectural evolution of Soomla. The Soomla[6] framework is an open-source framework for in-app purchases in Android. The second two case studies are related to the open source project Apache CXF[7] which is a framework for implementing Web services. This is implemented in Java and is built around a central Interceptor Chain which is used to handle incoming and outgoing messages on the client- and on the server-side. The second difference among these case studies is related to how the approach was applied. Specifically:

- In the first Case Study 1, described in Section 6.1, we used our approach to add support for a new payment method to the Soomla framework (v3.3). This case study shows how the approach would be really put into practice, as we have both planned the evolution and implemented the changes accordingly.
- In Case Study 2, described in Section 6.2, we retraced the architectural changes between version 3.2 and version 3.3 of Soomla using our approach. This case study, and the previous one, have facilitated the evaluation of the applicability of the approach when dealing with medium-size projects. This is specially relevant to determine whether the effort of applying the approach is negligible when compared to the development effort.
- In our third case study, described in Section 6.3, we used our approach to retrace the changes from Apache CXF Version 2.6 to Version 2.7. This and the following case study enable the evaluation of the applicability of the approach in a non-trivial system implemented by others, as Apache CXF is a larger size project, along with the effort needed when dealing with larger size projects.
- In the last case study, illustrated in Section 6.4, we retraced the changes from Apache CXF Version 2.7 to Version 3.0. This enables us to evaluate the approach when retracing the evolution between two major different versions, as opposed to the third case study, where we studied rather incremental changes.

[5] http://www.eclipse.org/xtend/
[6] http://soom.la/
[7] https://cxf.apache.org/

6.1. *Case Study 1: Soomla v3.3 Implementation of a New Custom Payment Provider for Payment Via Carrier*

In this case study we implemented a new payment option for the Soomla framework that supports payment via a custom local payment provider that offers payment via a Restful API. Soomla is a framework that, in newer versions, helps implementing In-App purchases in Android, iOS, and Unity applications but was originally developed purely for Android. We first documented the ADD to add this functionality (see Figure 5) and then used our approach:

Name	Implementation of the Restful Billing Provider
Group	Soomla Billing
Issue	Currently no support for payment via our custom Restful webservice exists.
Decision	Implement the restful biling using the new interfaces.
Positions	1) Implement a custom payment-provider based on the new billing abstractions and the restful web service API provided by the payment provider. 2) Do not support the custom payment via this provider.
Arguments / Implications	1) This requires a new RestfulBilling component (and a connector to the BillingProvider) in which we implement the interfaces defined in the BillingProvider and map them to the existing Restful API. 2) As the necessary effort is rather low and we can provide additional options to the customers.
Related decisions	Google-independent Billing Reimplementation of the Google Billing provider Implementation of the Amazon Billing provider

Figure 5: Architectural decision to add support for our custom payment provider using a Restful service

```
Component RestfulBilling
    consists of
      Package(root.com.soomla.store.billing.restful) or
      {
            Package(root.com.soomla.store.billing.restful)
            and
            InstanceOf(root.com.soomla.store.billing.IIabService)
      }
```

Figure 6: Architecture abstraction specification for the RestfulBilling component.

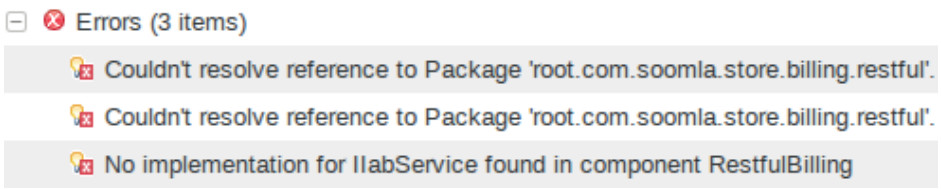

Figure 7: Consistency report created in the Generate step for the new RestfulBilling Component

- Transform step: We first used the addComponent-transformation to create a new component named RestfulBillingProvider with a new architecture abstraction specification (shown in Figure 6) and only one connector to the BillingProvider component. This brought us to *t+1* in Figure 2.

- Generate step: Based on this updated architecture specification we used our Reconstruction Tool to generate a UML component view of the Soomla's architecture and obtained a consistency report that lists the issues the tool detected after the architecture specification was transformed. In this case, these issues were that the new component (RestfulBillingProvider) and the new connectors were not present in the source code. More specifically that no package com.soomla.store.billing.restful existed and that no classes existed in this package that implemented the IIabService interface which is defined by the BillingProvider (shown in Figure 7). This brought us to *t+2* in Figure 2
- Implementation step: In this step we tackled the issues reported by the tool one by one and first implemented the code to provide the desired functionality by implementing the interfaces defined by the BillingProvider. The most important of these interfaces is the IIabService which is the new provider-independent interface used throughout the Soomla store. As required in the architecture abstraction specification, we placed our code in the package com.soomla.store.billing.restful. The completion of the implementation step brought us to *t+3* in Figure 2.
- Check for violations step: We then used the Reconstruction Tool to check for any remaining inconsistencies (*t+4* in Figure 2). However all issues from the Generate step were fixed and the evolution step was complete.

6.2. Case Study 2: Soomla Store version 3.2 to 3.3

For this case study we retraced the architectural changes for the Soomla Store from version 3.2 to version 3.3. Soomla is a framework that, in newer versions, helps implementing In-App purchases in Android, iOS, and Unity applications but was originally developed purely for Android. For this reason, the Soomla framework in version 3.2 only supported one payment method: Google Play Store. However, the requirement to also support payment via Amazon arose. Since payment via Google was so far "hardcoded into the framework, this required to change Soomla's architecture to support multiple payment providers and thus introducing an abstract representation of payments and payment providers instead of using the ones provided by Google. Soomla's architecture and the changes applied in this case study are shown in Figure 8.

ID	**Evol. Step**	**Architectural change**	**Arch. transformations**
1	1	Adding new Google-independent BillingProvider component	addComponent addConnector (used multiple times - for all components that used have a connector to the old Billing component
2	1	Removing the old – Google-specific Billing component	deleteComponent deleteConnector (used multiple times - to delete all connectors to and from the old Billing component)
3	2	(Re-)Adding support for GooglePlayBilling to the new BillingProvider	addComponent addConnector (to create a connector between BillingProvider and GooglePlayBilling)
4	3	Adding the new AmazonBilling component that integrates with the new BillingProvider	addComponent addConnector (to create a connector between BillingProvider and AmazonBilling)

Table 2: List of the architectural changes performed for the architecture evolution from Soomla v3.2 to Soomla v3.3.

Therefore, using our approach, we again performed multiple evolution steps. During the first evolution step, first, we used our approach for the ADD to introduce a provider independent payment representation. In the second evolution step, we employed our approach for the introduction of the new Google payment provider and in the third evolution step we carried the ADD to add support for payment via Amazon. In the following we provide details for each one of these steps. Figure 11 shows the architecture abstraction specification for the new components. We have summarized all necessary architectural changes for this version change in Table 2. In the following, these three steps are described in detail.

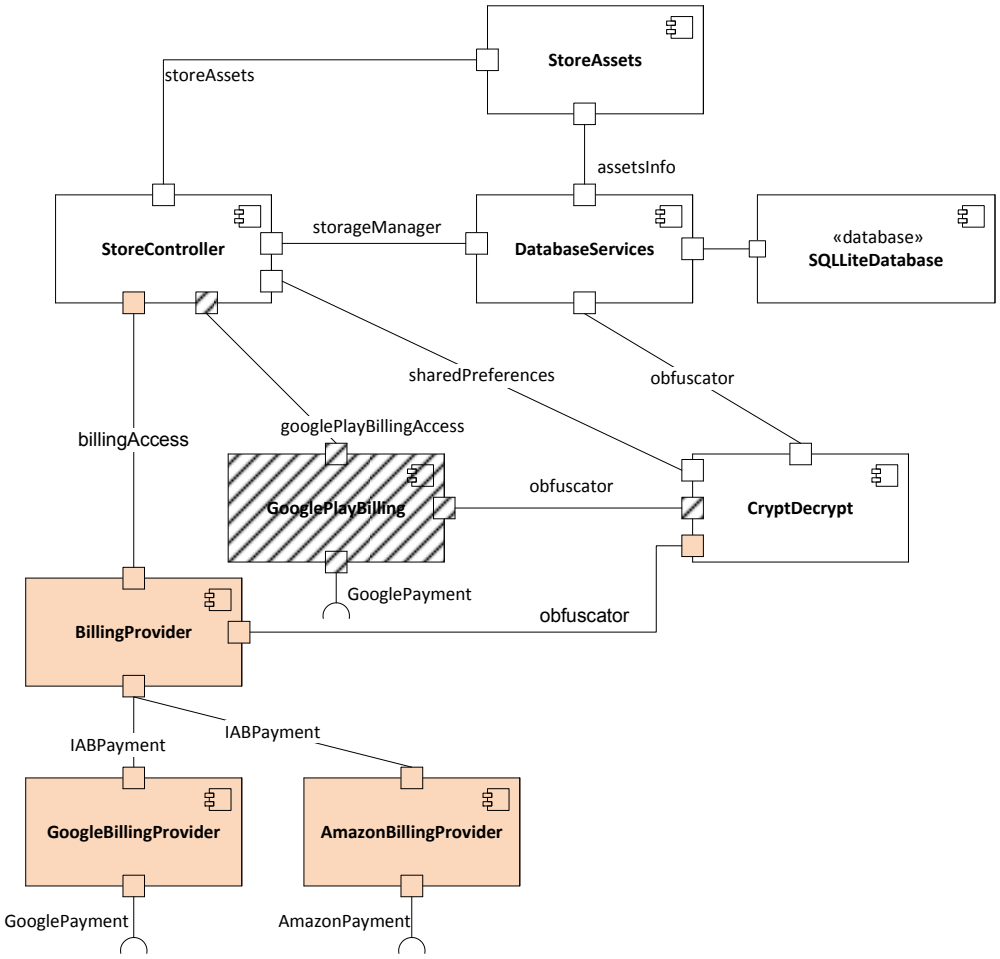

Figure 8: Soomla Architecture Overview showing the architecture for version 3.2 and the changes for version 3.3. Remove elements have a hatched background and new elements have a colored background.

1. During the first evolution step, we documented the ADD to replace the Google-specific billing component with a provider-independent billing component (see Figure 9) and then enacted the iterative process described in Section 3 as follows:

   - Transform step: We first used the addComponent-transformation to create a new component named BillingProvider with a new architecture abstraction specification. Then, we used the deleteConnector-transformation and the addConnector-transformation to replace all connectors to the old Billing component with connectors to the new BillingProvider component before using the deleteComponent-transformation to remove the old Google-specific Billing component.
   - Generate step: Based on this updated architecture specification we used our Reconstruction Tool to generate a UML component view of Soomla's architecture and obtained a consistency report that lists the issues the tool detected after the architecture specification was transformed. In this case, these warnings were that the new component BillingProvider and the updated connectors were not present in the source code and that the implementation of the old Billing component still existed in the source code of the system.
   - Implementation step: Similarly to the case studies presented in the previous sections, the implementation step (of this evolution step) follows and was based on the ADD, the UML component view, and the list of issues. As we are studying an already existing system, instead of imple-

Name	Google-independent Billing
Group	Soomla Billing
Issue	Currently only supports payment via Google.
Decision	Restructure the architecture and introduce an abstract billing representation.
Assumptions	In the future other billing providers should be supported.
Positions	1) Implement support for other providers using the current architecture and somehow wrap the current Google Billing provider. 2) Restructure Billing and introduce a provider independent Billing component with abstract billing interfaces and update the connectors accordingly.
Arguments / Implications	1) This does not cost any effort now but implementing other billing providers will require substantial changes to all of the framework and might require "ugly" workarounds. 2) This requires changes to all classes that access billing but will ease implementation of new providers in the future.
Related decisions	New Google Billing Provider

Figure 9: Architectural decision to implement provider independent billing in the Soomla framework

menting the new component, we updated the Soomla source code from Version 3.2 to Version 3.3 which concluded our implementation step and brought us to time (t+3) in Figure 2.

- Check for violations step: We again used the Reconstruction Tool to identify any remaining inconsistencies. All issues from the Generate step were fixed and the first evolution step was finished.

Name	Reimplementation of the Google Billing Provider
Group	Soomla Billing
Issue	Introduction of an independent billing representation requires a reimplementation of Google Billing
Decision	Implement Google Billing using the new interfaces
Assumptions	Other payment providers are already planned.
Positions	1) Reimplement based on the new BillingProvider component. 2) Revert the changes to introduce a provider-independent billing and go back to support only Google.
Arguments / Implications	1) This requries a new GoogleBilling component (and a connector to BillingProvider) in which we implement the interfaces defined in the BillingProvider component. 2) It is not really an option as other payment providers shall be implemented in the near future and their implementation is almost impossible with the old architecture.
Related decisions	Google-independent Billing

Figure 10: Documented architectural decision to re-add support for Google Play Billing

2. During the second evolution step, we documented the ADD to (re-)add support for payment via Google Play (see Figure 10) and then enacted the iterative process described in Section 4 as follows:

   - Transform step: We used the addComponent-transformation to create a new component called GoogleBillingProvider which is connected to the BillingProvider component. Figure 11 shows the architecture abstraction specification for the new BillingProvider and GooglePlayBilling components.

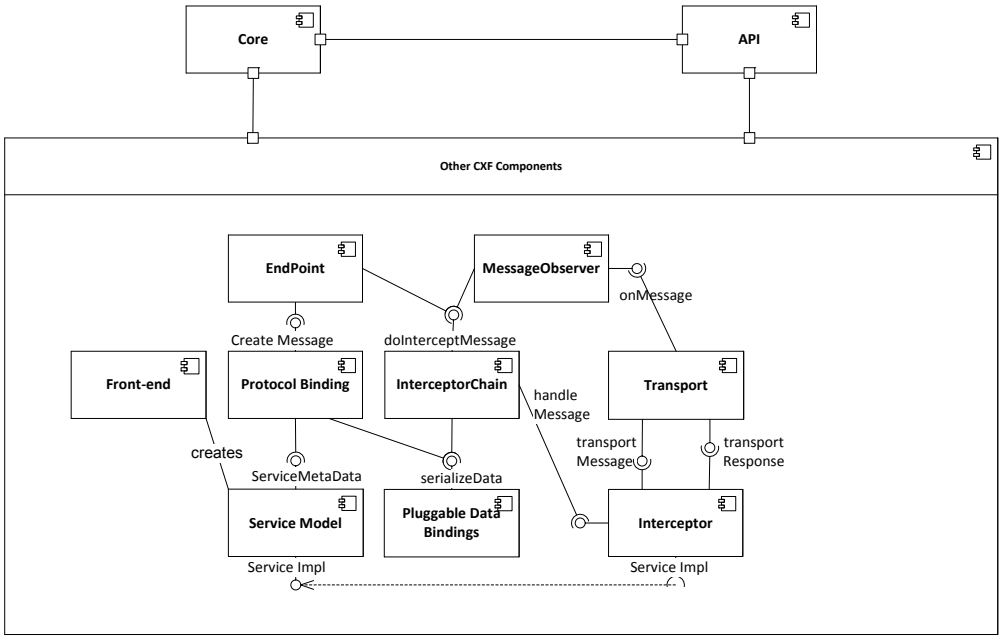

Figure 11: Architecture abstraction specification for the new provider independent Soomla Billing component and the new components that implement the payment providers for Google and Amazon

- Generate step: Based on this updated architecture specification we used our Reconstruction Tool to generate a UML component view of Soomla's architecture and obtained a consistency report that lists the issues the tool detected after the architecture specification was transformed. In this case, these warnings were that the new component and the new connectors were missed in the source code.
- Implementation step: Again, as we are studying an already existing system, instead of implementing the new component, we updated the Soomla source code by adding the module for payment via Google, which is developed in a separate repository since Soomla supports multiple payment providers. In fact, both currently supported payment providers are developed as separate modules and the Soomla framework itself does not contain any payment provider-specific code.
- Check for violations step: As before, we used the Reconstruction Tool to identify any remaining inconsistencies. All issues from the Generate step were fixed and the evolution step to (re-)add support for payment via Google Play was finished.

3. During the third evolution step, we documented the architectural decision to add support for payment via Amazon (shown in Figure 12). In this ADD we noted only one position – the one to implement support for payment via Amazon using the new Billing architecture. We also documented the other related decisions, in this case the decision to utilize the provider-independent billing architecture and the decision to re-implement the payment via Google using this architecture. In the case of needing to revise the question on how to implement this AmazonBillingProvider, the related questions probably also need to be revised. Then we enacted the iterative process described in Section 4. As this consisted of exactly the same steps as in evolution step 2, we skip the details at this point. We have already shown the architecture abstraction specification for the Amazon-specific payment component in Figure 11 and its integration into the architecture in Figure 8. Like the Google-specific code, the code for payment via Amazon is also developed in a separate module that was added in the implementation step. No inconsistencies were found during the check for violations step and once we finished with this evolution step, we had also completed the architecture evolution of Soomla from version 3.2 to version 3.3.

Name	Implementation of the Amazon Billing Provider
Group	Soomla Billing
Issue	Currently no support for payment via Amazon exists.
Decision	Implement Amazon Billing using the new interfaces
Positions	1) Implement Amazon-payment based on the new billing abstractions and the Amazon API.
Arguments / Implications	1) This requries a new AmazonBilling component (and a connector to BillingProvider) in which we implement the interfaces defined in the BillingProvider component using the Amazon API.
Related decisions	Google-independent Billing
	Reimplementation of the Google Billing Provider

Figure 12: Documend architectural decision to add support for payment via Amazon.

ID	Architectural change	Affected view	Arch. transformations
1	Added support for service discovery	architecture overview	addComponent addConnector
2	Added support for sending messages using UDP protocol	transport view	addComponent addConnector
3	Added support for sending SOAP messages using the UDP protocol	transport view	addComponent addConnector
4	Added support for asynchronous messages over HTTP	transport view	addComponent addConnector (used twice)
5	Partial support for JAX-RS v2.0 (support for JAX RS v1.1 already existed)	frontend view	updateAbstraction Specification

Table 3: Overview of all architectural changes from Apache CXF Version 2.6 to Apache CXF Version 2.7

6.3. Case Study 3: Evolving from Apache CXF 2.6 to Apache CXF 2.7

In this case study we used our approach to retrace the changes from Apache CXF Version 2.6 to Version 2.7. In total this version change raised the number of lines of Java source code from about 480.000 to more than 513.000 and consisted of 867 modified classes, 121 new classes, and 7 removed classes. For this case study we created three architectural component views of the CXF architecture using our Architecture Abstraction DSL. These views consist of a high-level overview of the complete architecture as well as two detailed views, one for the architecture of CXF transports and one for the architecture of CXF front-ends. While a CXF transport implements a specific protocol that is used to send messages, CXF front-ends define and implement the different types of supported services like e.g. JAX-WS, JAX-RS.

In Figure 13 we show a detailed view for the architecture of Apache CXF transports (showing only a subset of all supported transports). In Apache CXF, transports are used to abstract from the protocols used to send messages from the client to the server and back. Each transport has to provide implementations for the following three concepts: Conduits represent a channel for sending a message, Destinations represent the location of a service (a ServiceEndpoint), and TransportFactories are used to obtain transports for specific URLs.

As we studied the evolution of Apache CXF from Version 2.6 to Version 2.7 we discovered a number of architectural changes to the different views which are summarized in Table 3. In order to illustrate our approach in detail, we picked the change with ID 2, that is, the architectural changes necessary to add support for the UDP protocol. The implementation of a new protocol might be necessary when using Apache CXF in a restricted environment or when a new protocol should be supported. In Apache CXF this was the case for Version 2.7 where a new UDP transport was implemented.

In this case, we study the architectural evolution of the given architecture when the ADD to support

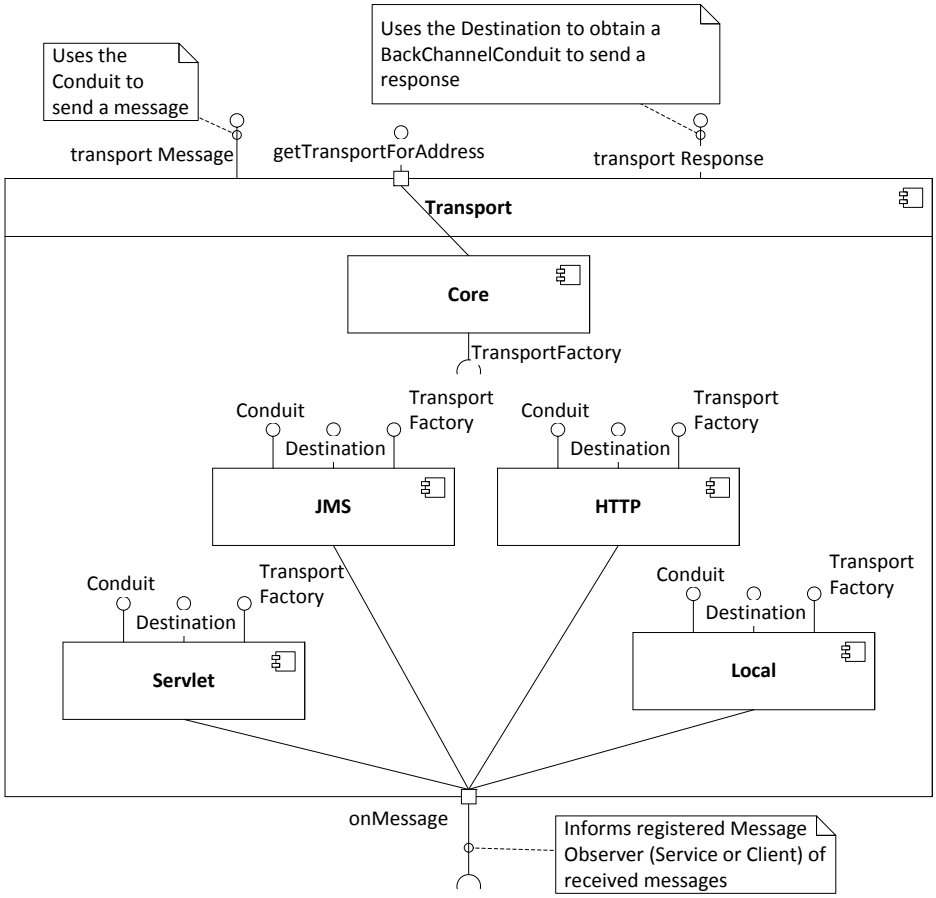

Figure 13: A simplified view of the architecture of Apache CXF transports

a new protocol has been taken. As we focus on studying the architectural evolution, we first created an architecture specification for Apache CXF Version 2.6 which we used as the basis for our approach (the architecture specification at t in Figure 2). Our approach assumes that this documented architecture already exists. If no initial architecture documentation exists, the semi-automatic architecture reconstruction approach (described in Section 5) or other existing architectural reconstruction approaches can be used to obtain an architectural description of a system in its current state. In the following, it is described how an iteration over the evolution step presented in Section 4 was carried out:

- Transform step: After the ADD to implement this new protocol was documented (see in Figure 14), we executed the transform-step of our approach. In our example, we first used the addComponent transformation to add a component named UDP to the architecture specification for the CXF transports. We exemplarily show the QVT-o transformation for adding a component in Figure 4. All other QVT-o used throughout the case studies can be found in Appendix A. The added component with its architecture abstraction specification is shown in Figure 15. We use Eclipse Launch configurations in conjunction with the QVT-o transformations to record all the architectural changes carried out during a transformation step (see Appendix B for an example of the launch configuration for adding the new UDP component). We defined this architecture abstraction specification in the following way: using the Architecture Abstraction DSL, we first specified that source code for this component should be part of the Package org.apache.cxf.transport.udp (specified by the UDP component's first Package rule in Figure 15) and second, we defined that this package and, thus, the component needs to contain an implementation of the interfaces ConduitInitiator (which is the interface for the Transport Factory),

Name	Add UDP Transport Support
Group	Transports
Issue	If and how should support for communication via UDP be implemented in Apache CXF?
Decision	UDP support should be implemented as a transport using the Apache MINA library.
Assumptions	UDP support will be integrated as a CXF Transport and is requried by a significant number of users.
Positions	1) Add a new UDP transport component which has a connector to the Transport core and implement it using Apache MINA. 2) Add a new UDP transport component which has a connector to the Transport core and implement it using plain old Java. 3) Do not support UDP.
Arguments / Implications	1) Apache MINA reduces the implementation effort for the UDP transport but introduces an additional external dependency. 2) It does not introduce the dependency but requires more written source code – which costs more time and might introduce bugs the could be avoided using a tested library. 3) The demand for UDP transport support is high enough to warrant the effort.
Related decisions	Design the Apache CXF Transport Architecture
Notes	This feature is planned for Apache CXF 2.7 which is scheduled for release in June 2013.

Figure 14: Documented architectural decision to implement UDP transport support for Apache CXF

Conduit, and Destination (specified by the UDP component's second Package rule and the InstanceOf rules in Figure 15). Therefore, the consistency checks of our Reconstruction Tool can evaluate whether the transport fulfills all the requirements that Apache CXF defines for its transports, that is, it enables architects to check all the constraints imposed by the style (see Section 3.1). At this point, we finished the transform-step and reached the architecture specification shown in Figure 2 at time (*t+1*).

- Generate step: Based on this architecture specification we used our Reconstruction Tool to generate a UML component view similar to the one already shown in Figure 13. However, the new component view also contains the new architectural component for the UDP transport. In addition, our Reconstruction Tool provides a consistency report that lists the issues the tool detected after the architecture specification was transformed. Initially, this report, as shown in Figure 16, consisted of multiple errors stating that no source code elements could be detected that adhered to the different elements in the architecture abstraction specification of the UDP component. This happens because no package org.apache.cxf.transport.udp existed in the source code yet and no realization of the interfaces Conduit, Destination, and ConduitInitiator could be found for the UDP component. After the generate step, we have successfully created the UML component view as shown in Figure 2 at time (t+2).

- Implementation step: Based on the ADD, the UML component view, and the list of issues, the implementation step (of this iteration) follows. As we are studying an already existing system, instead of implementing the new component, we updated the Apache CXF source code from Version 2.6 to Version 2.7 which concluded our implementation step and brought us to time (t+3) in Figure 2.

- Check for violations step: In this step we used the Reconstruction Tool to check whether the aforementioned issues still remained. While the Reconstruction Tool did not report any issue regarding the UDP component itself, it reported that there was a connection between the Core and the UDP component in the source code that was not covered in the architecture specification. After a short investigation, we decided that the detected violation was a violation in the ADD that occurred because

```
Component Core
        consists of
        {
                Package (org.apache.cxf.transport, excludeChildren)
                or
                Package (org.apache.cxf.transport.common)
        }

Component UDP
        consists of
        Package (org.apache.cxf.transport.udp)
        or
        {
                Package (org.apache.cxf.transport.udp)
                and
                {
                        InstanceOf (org.apache.cxf.transport.ConduitInitiator)
                        or
                        InstanceOf (org.apache.cxf.transport.Conduit)
                        or
                        InstanceOf (org.apache.cxf.transport.Destination)
                }
        }

//[…]
```

Figure 15: Excerpt of the architecture specification showing the Core component of the transport view as well as the new architectural component that was added during the first transformation step of our case study with syntax highlighting for reported inconsistencies

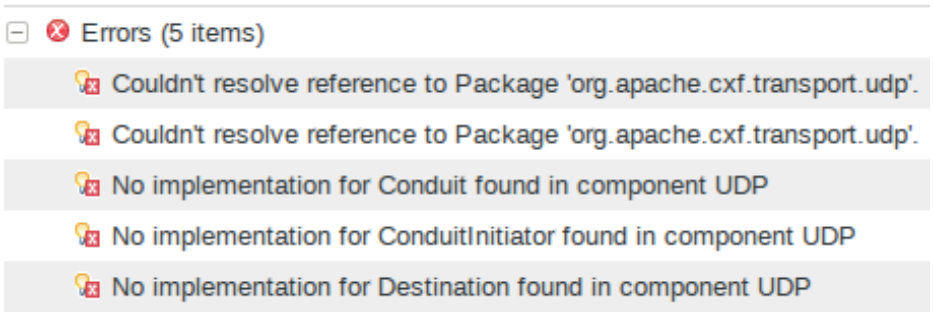

Figure 16: Consistency report for the new architectural component that was added during the first transformation step in our case study (as shown in Figure 15)

a connector between those two components in the architecture was necessary.

We started a new iteration of our approach which led us back to iterate over the evolution step, applying every one of the identified steps Figure 2. During this second iteration, we first updated the ADD and then, in the transformation step, used the addConnector transformation (see Appendix A, Figure A.23) to create a new connector in the architecture specification that connects the Core component with the before added UDP component. After this, we skipped the implementation step, because it was not necessary to change the code, and again used the Reconstruction Tool to check for issues in the check for violations step. As no issues were reported any more, we finished this architecture evolution step.

6.4. Case Study 4: Apache CXF 2.7 to Apache CXF 3.0

In this case study we did retrace the version change from Apache CXF 2.7 to 3.0 using our approach. In this version change, the number of lines of source code grew to about 560.000 and consisted of 1224 modified, 296 new, and 111 removed classes. Figure 17 shows the high-level architectural component view

of Apache CXF 2.7 before any changes were performed. This major update contained a number of changes to CXF that affect the implemented architecture. Among these changes are two new supported transports for which we performed the same architecture transformation steps as for the new UDP transport in the previous case study.

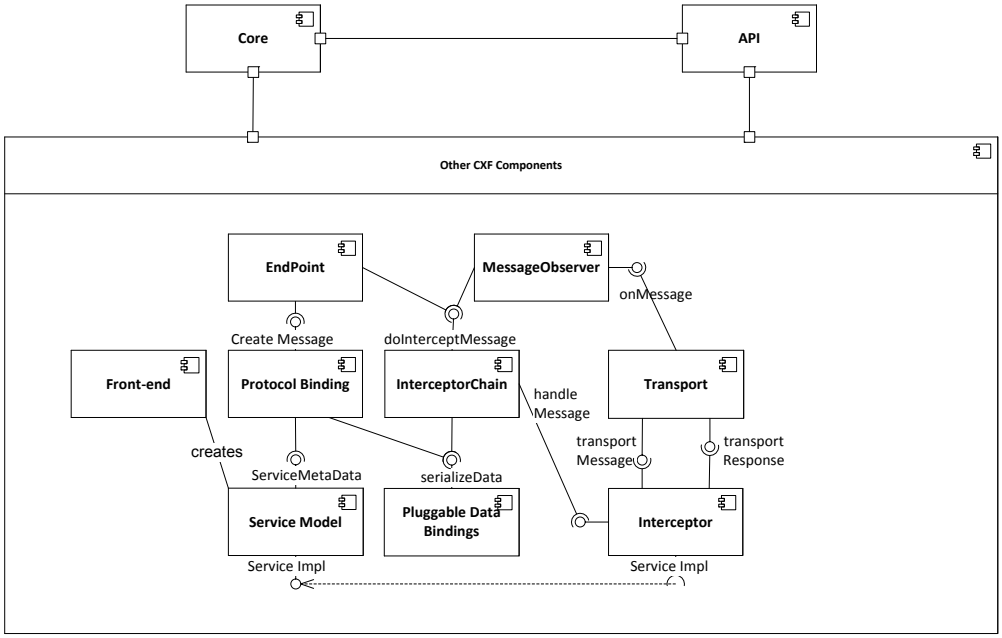

Figure 17: Architecture overview of Apache CXF 2.7

Name	Core and API components merger
Group	Core Architecture
Issue	The other components all depend on the components CORE as well as API.
Decision	Merge the two components.
Assumptions	As the core component will contain the code from the API no changes to components should be necessary.
Positions	1) Keep them separated. 2) Merge the two components.
Arguments / Implications	1) While the separation of CORE and API has its pros - especially a separation of interfaces and implementations -, it also introduces a high-number of dependencies and makes things more complicated. 2) This will result in a big core component but makes the architecture simpler.
Related decisions	Core Architecture Design
Notes	Scheduled for CXF 3.

Figure 18: Documented architectural decision to merge the Components Core and API

During this version change, two central components of the system, the CORE and the API components, have been merged into a single CORE component and the WSDL-related functionality has been moved into a component of its own called WSDL. Therefore, using our approach, we performed two evolution steps:

1. During the first evolution step, we documented the ADD to describe that the two components were merged (see Figure 18). In this ADD, we documented the problem, the outcome of the decision, named

different possible solutions and arguments for these solutions, as well as related this decision to other affected decisions. Then we enacted the iterative process described in Section 4 as follows:

- Transform step: We first updated the architecture abstraction specification of the CORE component by applying the *updateAbstractionSpecification* transformation that is shown in Appendix A (Figure A.21). This merged the architecture specification of the API component and the CORE component. This caused the automated consistency checks of our tool to report an architectural inconsistency as the architecture abstraction specification of the components now overlapped. We then applied the deleteComponent (see Appendix A, Figure A.22) transformation to delete the API component which solved the aforementioned architectural inconsistency but also required us to apply deleteConnector (see Appendix A, Figure A.24 and Figure A.25) for every connector that expressed a dependency to the API component. Then we searched the components whose connectors had been deleted to replace them with new connectors that expressed the dependency on the modified CORE component. However, all of the components already expressed this dependency so that no new connectors were necessary.
- Generate step: Based on this updated architecture specification we used our Reconstruction Tool to generate a UML component view of Apache CXF and obtained a consistency report that lists the issues the tool detected after the architecture specification was transformed. This list includes one message for each class that should be moved from the API to the CORE components.
- Implementation step: Similarly to the case study presented in the previous section, the implementation step (of this evolution step) follows and was based on the ADD, the UML component view, and the list of issues. As we are studying an already existing system, instead of implementing the new component, we updated the Apache CXF source code from Version 2.7 to Version 3.0 which concluded our implementation step and brought us to time (t+3) in Figure 2.
- Check for violations step: We again used the Reconstruction Tool to identify any remaining inconsistencies. While the list of issues from the Generate step were all fixed, a number of classes were identified as being misplaced. These were the classes that provide the WSDL functionality in Apache CXF which were separated from the new Core component and moved into a separate WSDL component. Until this point, we had not updated our architecture description with respect to this change. In order to deal with this change, we went back and performed the second transformation step described in the following.

Name	Simplify the Core component
Group	Core Architecture
Issue	After merging of Core and API, Core has grown very big.
Decision	A new WSDL component will be split of the Core and hold the WSDL functionality.
Positions	1) Revert the decision to separate the Core and API components. 2) Move the WSDL related code to a separate WSDL component.
Arguments / Implications	1) This has the drawbacks already described in the Core and API components merger decision. Especially the overall number of connectors in the architecture would increase dramatically. 3) Splitting of the WSDL component reduces the Core's complexity and does not introduce the same amount of complexitiy as position 1.
Related decisions	Core and API components merger

Figure 19: Documented architectural decision to separate the WSDL related functionality from the Core component

2. During the second evolution step, we first documented the ADD shown in Figure 19 to describe that the WSDL-related functionality was moved to a new component and then carried out the evolution as follows:

- Transform step: In this step, we used the addComponent and the updateAbstractionSpecification transformations to create the new WSDL component (shown in Figure 20) according to the abstraction specification and to remove those classes implementing the new component from the abstraction specification of the CORE component, respectively.
- Generate step: we regenerated the UML component model and, as we are applying the process to an existing system, we skipped the implementation step and directly checked for violations.
- Check for violations step: Once the above-mentioned changes to the architecture were carried out, no more violations were detected by the Reconstruction Tool and we finished the retrace of the version change from Apache CXF 2.7 to 3.0.

6.5. Effort Estimations for the Four Cases

As addressed by RQ3, a critical question for our approach is whether the extra time needed to put it into practice is negligible with regard to the total time needed for the evolution of a system. In order to answer this question, we have collected data from each case study. In particular, we have collected precise data to characterize the evolution in the four cases studied (see Table 4). The evolution in Case Study 1 was a new development, performed by us, whereas Case Studies 2-4 are retraced evolutions performed by others.

Case Study	**Old Version**	**New Version**	**Number of Lines of Changed Java Code**	**Number of Changed or New Java Files**	**Number of New Java Files**	**Number of Deleted Files**
Case study 1	Soomla 3.3	Soomla 3.3 with Payment Provider	213	2	2	0
Case study 2	Soomla 3.2	Soomla 3.3	2229	24	15	13
Case study 3	CXF 2.6	CXF 2.7	61551	869	162	9
Case study 4	CXF 2.7	CXF 3	182239	1469	545	259

Table 4: Characterization of Evolution in the Four Cases

In Table 5 we report the exact efforts the first author, who was experienced in depth with our approach and the two systems in focus, needed to apply our approach in staff hours. We expect that a professional developer of a particular system applying our approach is more familiar with that system than we are, but also that some additional effort is required, because of the unfamiliarity with our approach. So the numbers in column Effort for Our Approach in Staff Hours might increase slightly when applied by others.

Using our approach requires some initial effort to create the architecture documentation using our Architecture Abstraction DSL. This is also reflected in Table 5, where the time required for our approach is higher for Case Study 3 than Case Study 4, although in the later case, substantially more changes to the architecture are made. Once the architecture documentation already exists (as in Case Study 1 and Case Study 4), the remaining effort comes from encoding the architecture changes applying the QVT-o transformations we already provide. This serves the purpose of explicitly documenting the architectural changes while reducing the effort for actually changing the architecture description to run the transformations.

```
Component RTWSDL
        consists of
        {
                Package (root.org.apache.cxf.wsdl)
                or
                Package (root.org.apache.cxf.wsdl11)
        }
        and not
        Package (root.org.apache.cxf.wsdl.http)
```

Figure 20: The WSDL component that now holds WSDL relevant functionality in Apache CXF 3.0

While we have no precise numbers, except for Case Study 1, on how long the evolution of the different versions of Soomla or CXF took, given the rather low numbers of staff hours it needs to apply our approach for these substantial changes, it is rather likely that applying our approach takes just a very small fraction of the staff hours required for the evolution itself. For a very rough comparison we have used the numbers estimated by Cocomo II [7] that would be needed for writing the changed lines of code (in staff hours). To be on the save side, we were very careful to use very conservative parameters for Cocomo estimations (in terms modified vs. new lines of code, as well as parameters such as assuming nominal *software understanding*, *unfamiliarity* as mostly familiar, low *integration requirements*, and considerable *documentation* begin available). For Case Study 1, only the development effort spent by the first authors was 40 hours. He estimates another 40-60 hours would be needed to turn this code into production-ready code. The Cocomo II estimates are 66 hours, which confirms that the Cocomo II are roughly fitting, but are rather conservatively estimated.

The results in Table 5 indicate that our approach requires only a minimal amount of time compared to these estimates. Other reported lines of code based estimation of development productivity come to estimates in a range substantially below these numbers. For instance, the productivity rates for Java reported by Phipps [60] range between 0.97 to 2.74 LoC per staff hour, which would lead to substantially lower numbers for the effort comparison. Jones [36] reports similar 2.13 LoC per staff hour for C++ (for one specific PBX project) and rates C++ at the same language level as Java in terms of LoC required per function point. Although our resulting figures are based on rather rough estimations, they provide us with an initial indication that the benefits of our proposal outweigh in the four cases the extra workload it requires.

Case Study	**Effort for Our Approach in Staff Hours**	**Writing the new & modified LoCs in Staff Hours**	**Effort Comparison in %**
Case study 1	0.20	40.30	0.49%
Case Study	**Effort for Our Approach in Staff Hours**	**Cocomo II Estimate for Writing the Changed LoCs in Staff Hours**	**Effort Comparison in %**
Case study 2	0.20	66.00	0.30%
Case study 2	1.20	1039.50	0.10%
Case study 3	7.03	9273.00	0.03%
Case study 4	1.80	74134.50	0.00%

Table 5: Effort of our Approach Compared to Cocomo II Estimations in Staff Hours

Please note that in Table 5 we separated Case Study 1 from the others because, while we provide estimates for Case Studies 2, 3, and 4, we report the actual time we required for writing the new source code for Case Study 1. While our lines of code per staff hour are higher, we did not create a production ready implementation but only a prototype and thus the time reported for this Case Study cannot easily be compared to the other reported times.

7. Discussion

7.1. Main Contributions in Relation to the Research Questions

As was stated in Section 1, three research questions have driven the design of our approach. In the following, we provide answers to these questions based on the conclusions drawn from our design science study and from the four case studies we have carried out.

RQ1: *Is it possible to effectively combine enforcing the integration of software architecture and source code to avoid architectural erosion and architectural drift with architecture-guided evolution planning? If yes, how can it be achieved?*

Our approach provides architects with assistance to plan the evolution. The case studies have illustrated that architects can plan the evolution by setting different evolution steps. Each one of these evolution steps conforms to one ADD and is automated by applying QVT-o transformations. That is, the architecture abstraction specification is updated accordingly. Developers have to carry out their coding task by using

such ADD as a coding guideline, as well as the architecture documents automatically generated using the architecture abstraction specification. This planning of the evolution offers several advantages:

- Traceability links between the architectural decisions and the affected architectural elements are automatically recorded thanks to the documented QVT-o transformations. This means that architects can retrace any change done throughout the history of the system.
- Traceability links between the architectural elements and the source code are also maintained. This enables architects and developers to be able to automatically relate code changes to specific architectural changes.
- Finally, changes in the source code are properly linked to ADDs, so that developers can understand why the code became structured the way it is.

In addition to the benefits for planning the evolution, the approach also enforces the integration of software architecture and source code to avoid architectural erosion and architectural drift. The aforementioned traceability links between the architectural elements and the source code are the main means in our approach to handle this kind of integration. For the support provided by such traceability links we have established objectively quantifiable evidence in our previous work. In a series of four controlled experience [34, 33, 35, 32] we have shown that human analyst performance in software architecture understanding and change impact analysis during evolution is significantly improved, if traceability links are used as a means for the integration of software architecture and source code.

We can also highlight some anecdotal evidence from the case studies to illustrate how the integration of software architecture and source code has helped us in these case to spot inconsistencies early on. In the case studies it was shown that in each evolution step, developers and architects were required to check whether the code conforms to the architecture as planned, by using the support the Reconstruction Tool offers. As a result of this checking, during our case studies we have identified a number of discrepancies between architecture and source code. These inconsistencies do not constitute a problem of the approach, but highlight some of the problems the approach helps to identify early in the evolution process. For instance, in Case Study 3 (Section 6.3) it was shown that one of the constraints of the style imposes the implementation of specific interfaces. The use of the approach enables architects to establish these constraints and developers to check whether the code satisfies them. Therefore, this case study shows *the approach's ability to prevent architectural erosion* [61]. Another inconsistency discovered in Case Study 3 was a connector that existed in the source code, but not in the architecture documentation. This shows *the approach's ability to prevent architectural drift* [61] and is an example for the third type of consistency violation – as the architecture abstraction specification was missing, the connector, but neither the ADD nor the code, required changes. The inconsistencies reported in Case Study 4 at the end of the first evolution step stem from the problem that this case study retraces the steps of a real world application. The code changes between the two released versions could not be mapped to a single architectural decision and thus our approach reports inconsistencies when the code is advanced to the new version but the architecture has not been updated to this version yet. However, this also showcases the approaches ability to detect inconsistencies between the architecture and the source code.

As a result, we can conclude that the proposed approach supports enforcing the integration between software architecture and source code, to avoid architectural erosion and architectural drift, and at the same time provides automatic guidance to developers to carry out the required change tasks in each evolution step. That is, RQ1 can be positively answered.

RQ2: *If RQ1 can be positively answered, is it possible to apply the approach in realistic software projects?*

The case studies show the applicability of the approach for systems of different sizes and for different sizes of changes. While Soomla has about 5000 lines of source code, Apache CXF has somewhere close to 500.000 lines of source code. That is, these system significantly differ in size and complexity. In addition, different changes size and complexity of changes have been analyzed throughout the case studies. The number of lines of code that changed differ from 213 up to 299.292 lines of code, as shown in Table 5.

While in Case Study 1 we carried out an evolution step on a real-world system specifically to showcase the approach, the other three case studies demonstrate the approach's applicability for different types of architectural changes. The changes in the first two case studies only affect a limited number of architectural elements. While Case Study 1 and Case Study 3 contain architectural changes that do not affect the system as a whole, Case Study 2 and 4 do showcase substantial changes to these systems that affect a large part of the systems architectural elements. Furthermore, our Case Studies 2 and 3 demonstrate the applicability of the approach on different levels of abstraction. In Case Study 2 only a subsystem is changed, while in Case Study 3 multiple architectural elements throughout the whole system are affected by the changes.

Therefore, we have studied relatively small changes compared to the overall sizes of the systems up to changes which affected large parts of those two real systems. This enables us to clearly assess the applicability of the approach in four realistic cases. In the light of these preliminary results, RQ2 can also be positively answered, even though more evidence especially in industrial settings would be needed to generalize the results.

RQ3: *If RQ1 and RQ2 can be positively answered, is it possible to apply the approach with little extra effort compared to the effort roughly spent on the evolution of the software system in focus?*

RQ3 can be positively answered, as our effort estimations (see Section 6.5) clearly point in the direction that the effort required for our approach is minimal compared to the overall development effort necessary in the projects. While such quantitative estimations can give only a rough indication, and certainly more evidence is needed to get empirically sound numbers, the order of magnitude to which our approach requires less effort than the roughly estimated effort of the evolutions of Apache CXF and Soomla indicates that our results can likely be generalized.

It is worth noting that this answer has been developed only focusing on the extra effort required by the approach in order to offer a quantitative estimation. However, RQ3 can be also qualitatively answered by focusing on the effort that both developers and architects can save. First, according to Sillito et al. [63], one of the questions asked by programmers to carry out a change task focuses on finding points in the code that were relevant to the task at hand. Thanks to the coding guidelines and the architectural abstraction provided to the developers, we can shorten the time needed to find such points. Second, Brunet et al. [11] performed a longitudinal and exploratory study, which encompasses the analysis of 19 bi-weekly versions of four real system, to assess the architectural violations lifecycle and location over time. They detected more than 3.000 violations and noted that violations are usually solved in a different version to the one where they were introduced. This means that an extra effort is needed by developers to understand and locate where these violations are. However, as the case studies show, our approach supports software architects and software developers by automatically providing feedback on the consistency of the architecture and the source code through the Reconstruction Tool which includes reports about missing or misplaced source code as well as the automatic checking of constraints that software architects define for the architecture of the system. This way, our tool enables them to check automatically the constraints related to the evolution style. Third, as was stated in Section 3.2, Bratthall et al. [9] concluded that most of the interviewed architects considered that by using AK they could shorten the time required to perform change tasks. In the case studies, we have been shown that we are able to capture how and why specific evolution steps have happened as every evolution step was described by means of an ADD using ADvISE. This is important information that the Reconstruction Tool lacks if used on its own. Fourth, although the case studies are essentially lab experiments, they are all based on real world systems of considerable size and complexity. We aimed to follow realistic evolution scenarios as closely as possible in order to be able to generalize to a certain extent (see Threats to validity discussion) to a real-world setting. Therefore, the case studies, especially Case Study 1, have enabled us to experience how architects and developers would have worked with these systems, and we can vouch for the effort reduction, as presented in Section 6.5, in the sense that we aimed to follow the real evolution of the system as closely as possible. Finally, we think that the application of our approach is beneficial over the whole lifecycle of a software system. Our case studies show a relatively small impact on the necessary effort during the initial development of a software system, while our approach guides architects and developers during the evolution of a software system by providing traceability links between ADDs and source code as well as automatic consistency checking between architecture and code.

Therefore, while there is certainly room for improvement (see Section 7.2), it is important to show that it is at all possible to answer positively these three Research Questions.

7.2. Limitations and Threats to Validity of the Approach

It is also important to mention the limitations of our approach, as well as the main threats to validity, so that it can be evaluated from a practical point of view.

While Case Study 1 applies the approach as intended, a limitation of the later three case studies is that they retrace the architectural evolution of a system instead of using the approach during the evolution itself. Although we did a source code study of all source code versions and used other available material like e.g. change logs for our analysis, we cannot guarantee that the decisions we identified match the ones originally taken by the developers themselves. This constitutes a threat to validity for our evaluation. However, as the primary concern of these case studies is to demonstrate the applicability of the approach in a real world scenario, and while we cannot guarantee the perfect identification of the ADDs in these case studies, they still show its applicability.

In addition, this allowed us to study real systems of considerable size instead of showcasing the application on artificial examples. Conversely, in the first case study we did not retrace architectural evolution but applied our approach to perform architecture evolution by integrating an additional payment provider into the Soomla framework, in order to gain the view of a developer. This first case study allowed us to identify how valuable the information of the ADDs and consistency reports were to guide our implementation.

A limitation of our design science study based approach is that a number of measures have been combined in our approach to achieve the goals. For that reason we cannot precisely discern which parts of our approach have caused which parts of the observed effects. While for parts of our results substantial empirical evidence support our claims (e.g., the benefits for traceability links between architecture and source code, see Section 7.1), and hence the influence on the effects can be estimated, this is not the case for all aspects. This might also constitute a threat to validity regarding conclusion validity. That is, it is not clear whether all parts of our approach – to that exact extent – are needed to reach the same effects. Further studies are needed to get more evidence on those aspects.

A threat to validity is that all steps have been performed by the authors. While we showed that the approach is feasible and applicable, it is not clear yet, if it is applicable with the same effort by others. It is also not clear, if the experiences with four evolutions of two open source systems is enough to be able to generalize the results to any kinds of system, especially the applicability in industrial settings and other application domains would need to shown in future work.

One limitation of the approach, which it shares with all approaches that invest effort in the architecture documentation, is that it is not really applicable for small software systems as in these cases the source code often is enough for understanding the software architecture. However, this does not extend to middle and large scale systems where architecture documentation is necessary. Another limitation of the approach is that it focuses just on the static view of the system as only evolution operators such as add component, delete connector, etc, are currently supported. Moreover, it is worth noting that it has been designed for architectural changes. Therefore, changes at a lower abstraction level, such as adding a method to a class, are not supported by our approach. This constitutes one of the challenges that should addressed in our next future.

Another limitation, which we initially did not consider to be relevant, is that ADvISE and the Reconstruction Tool are currently two separate tools. A complete integration of those tools, which we plan in our future work, would improve the user experience for the software architect. For now they are only integrated by storing links to the documented ADD according the architecture transformations (written in QVT-o).

8. Conclusions and Future Work

In this article we have introduced an approach for the joint evolution of software architecture and source code. By integrating our software architecture reconstruction approach with AK and evolution styles, we were able to support the evolution of a system on an architectural level. It also lets the software architect

provides the software developer with guidelines on how to adapt the source code in order to conform to the changed architecture and to the constraints imposed by the architectural style. That is, to the best of our knowledge, we introduce the first approach that supports enforcing the integration between software architecture and source code, to avoid architectural erosion and architectural drift, and at the same time provides automatic guidance to developers to carry out the required change tasks in each evolution step. We were then able to perform automated consistency checking between the architectural specification and the source code. It is also noteworthy that all the different types of architectural changes, such as add component, delete component, etc., were automated by using QVT-operational, so that they can be easily traced and undone.

Moreover, we have also presented four case studies based on two real systems: Apache CXF and Soomla. They show the support for incremental changes to the architecture and the underlying source code provided by our approach. With this aim, it has been shown how the software architect can check whether the architecture specification and the source code are consistent. Furthermore, we provide the software architect with an iterative workflow to reach a state such that the documented architecture and the code have been changed according to the architectural decisions and are consistent with each other. As a result, we can clearly assess that the applicability in four realistic cases is given. Further, we found initial evidence that the effort required for our approach is very likely minimal compared to the overall development effort necessary in the projects.

In our future work we plan to automatically modify existing architecture specification guidelines between architectural component views and source code when the architectural component view is modified. Furthermore, we want to investigate if approaches for estimating the cost of architectural changes can be integrated with our approach. Finally, we aim to provide further evidence for our approach in controlled experiments and industrial settings.

Acknowledgments This research has been partly funded by the Spanish Ministry of Economy and Competitiveness and by the FEDER funds of the EU under the project Grant Vi-SMARt (TIN2016-79100-R) and by the Ministry of Education, Culture and Sport under the State Programme to Promote Talent and Employability in I+D+I, National Sub-Programme for Mobility belonging to the Spanish National Plan for Scientific and Technical Research and Innovation 2013-2016 (CAS14/00020).

Appendix A. QVT-O Transformations

In the following figures, the different transformations to apply the architectural changes are described. All of them have been implemented by means of QVT-operational and enable architects to retrace the architectural changes carried out during the evolution. As already described in Section 5, each figure depicts one of the following transformations:

- updateAbstractionSpecification. As Figure A.21 shows, this transformation modifies the architecture abstraction specification that defines how an architectural component relates to the source code.
- deleteComponent. Figure A.22 depicts how an architectural component is eliminated from the architecture abstraction specification.
- addConnector. Figure A.23 describes how a new connector between two architectural components can be created.
- deleteConnector: Figure A.24 and Figure A.25 show how a connector between two architectural components is deleted.

```
modeltype DSL uses
'http://www.univie.ac.at/cs/swa/component/architectureabstraction/ArchitectureAbstracti
onDSL';
transformation updateAbstractionSpecification
       (in dsl : DSL, in inputDSL: DSL , out outdsl : DSL);

main() {
       dsl.rootObjects()[DSL::Transformation]->
       map updateAbstractionSpecification(inputDSL.rootObjects()[DSL::Transformation]-
>asOrderedSet()
    ->first().map getComponent());
}

mapping  DSL::Transformation::updateAbstractionSpecification
       (in inputComp:DSL::ComponentDef): DSL::Transformation {
       result.name := self.name;
       result.components := self.components->
              map updateAbstractionSpecificationForComponent(inputComp);
}

mapping DSL::Transformation::getComponent () : ComponentDef {
       init {
       result := self.components->asOrderedSet()->first();
       }
}

mapping DSL::ComponentDef::updateAbstractionSpecificationForComponent
              (in inputComp:DSL::ComponentDef) : DSL::ComponentDef
//when {self.name = inputComp.name}
{
       init {
              result := self;
              if (self.name = inputComp.name) then {
                     result.expr := inputComp.expr;
              } endif;
       }
}
```

Figure A.21: QVT-o transformation for updating the architecture specification of a component

```
modeltype DSL uses
'http://www.univie.ac.at/cs/swa/component/architectureabstraction/ArchitectureAbstractionDSL';
transformation deleteComponentTransformation(in dsl : DSL, in inputDSL: DSL , out outdsl : DSL);
main() {
   dsl.rootObjects()[DSL::Transformation]-> map deleteComponent(inputDSL.rootObjects()[DSL::Transformation]
      ->asOrderedSet()->first().map getComponent());
}
mapping  DSL::Transformation::deleteComponent (in inputComp:DSL::ComponentDef): DSL::Transformation {
   result.name := self.name;
   result.components := self.components->excluding(inputComp)
}
mapping DSL::Transformation::getComponent () : ComponentDef {
   init {
      result := self.components->asOrderedSet()->first();
   }
}
```

Figure A.22: QVT-o transformation for deleting a component

```
modeltype DSL uses 'http://www.univie.ac.at/cs/swa/component/architectureabstraction/ArchitectureAbstractionDSL';
transformation addComponentTransformation(in dsl : DSL, in inputDSL: DSL , out outdsl : DSL);


main() {
dsl.rootObjects()[DSL::Transformation]-> map
                addConnector(inputDSL.rootObjects()[DSL::Transformation]->
                asOrderedSet()->first().map getComponent());
}


mapping  DSL::Transformation::addConnector (in inputComp:DSL::ComponentDef): DSL::Transformation {
                result.name := self.name;
                result.components := self.components->map addConnectorToComponent(inputComp);
}


mapping DSL::Transformation::getComponent () : ComponentDef {
                init {
                result := self.components->asOrderedSet()->first();
                }
}


mapping DSL::ComponentDef::addConnectorToComponent(in inputComp:DSL::ComponentDef) : DSL::ComponentDef
{
                init {
                result := self;
                if (self.name = inputComp.name) then {
                                var connector := inputComp.connectors->asOrderedSet()->first();
                                var addedConnector :=  self.connectors->including(connector);
                                result.connectors := addedConnector;
                } endif;
                }
}
```

Figure A.23: QVT-o transformation for adding a new connector to the architecture specification of a component

```
modeltype DSL uses
'http://www.univie.ac.at/cs/swa/component/architectureabstraction/ArchitectureAbstraction
DSL';
transformation deleteConnectorTransformation(in dsl : DSL, in inputDSL: DSL , out outdsl
: DSL);

main() {
       dsl.rootObjects()[DSL::Transformation]-> map
              deleteConnector(inputDSL.rootObjects()[DSL::Transformation]
              ->asOrderedSet()->first().map getComponent());
}

mapping  DSL::Transformation::deleteConnector (in inputComp:DSL::ComponentDef):
DSL::Transformation {
       result.name := self.name;
       result.components := self.components->map
              deleteConnectorFromComponent(inputComp);
}

mapping DSL::Transformation::getComponent () : ComponentDef {
init {
       result := self.components->asOrderedSet()->first();
       }
}

mapping DSL::ComponentDef::deleteConnectorFromComponent(in inputComp:DSL::ComponentDef) :
DSL::ComponentDef
{
       init {
       result := self;
       if (self.name = inputComp.name) then {
       var connector := inputComp.connectors->asOrderedSet()->first();
       var annotation := connector.annotation;
       // name based comparison necessary as these are different objects that
       // cannot be compared with equals
       var targetNames := annotation.oclAsType(ConnectorAnnotation)
                           ->targets()->collect(t| t.name);
       var connectorsToRemove := self.connectors ->
              select(c | c.annotation.targets()->collect(t|t.name)=(targetNames));
       var withoutRemovedConnector :=  self.connectors-(connectorsToRemove);
       result.connectors := withoutRemovedConnector;
       } endif;
       }
}
```

Figure A.24: QVT-o transformation for deleting a connector (part 1 of 2)

```
helper ConnectorAnnotation::targets() :
Sequence(PatternInstancePrimitiveTarget) {
	if(self.oclIsTypeOf(SimpleConnectorAnnotation)) then	{
		return self.oclAsType(SimpleConnectorAnnotation).targets;
	}endif;
	if(self.oclIsTypeOf(ShieldAnnotation)) then	{
		return self.oclAsType(ShieldAnnotation).targets;
	}endif;
		if(self.oclIsTypeOf(VirtualConnectorAnnotation)) then	{
		return self.oclAsType(VirtualConnectorAnnotation).targets;
	}endif;
		if(self.oclIsTypeOf(IndirectionAnnotation)) then	{
		return self.oclAsType(IndirectionAnnotation).targets;
	}endif;
		if(self.oclIsTypeOf(CallbackAnnotation)) then {
		var list : Sequence(PatternInstancePrimitiveTarget);
		list->append(self.oclAsType(CallbackAnnotation).target);

		return list;
	}endif;
		if(self.oclIsTypeOf(CompositeCascadeAnnotation)) then	{
		return self.oclAsType(CompositeCascadeAnnotation).targets;
	}endif;
		if(self.oclIsTypeOf(AggregationCascadeAnnotation)) then	{
		return self.oclAsType(AggregationCascadeAnnotation).targets;
	}endif;
	return null;
}
```

Figure A.25: QVT-o transformation for deleting a connector (part 2 of 2)

Appendix B. Launch Configuration for Executing a QVT-O Transformation

```
<launchConfiguration type="org.eclipse.m2m.qvt.oml.QvtTransformation">
<booleanAttribute key="org.eclipse.m2m.qvt.oml.interpreter.clearContents1" value="true"/>
<booleanAttribute key="org.eclipse.m2m.qvt.oml.interpreter.clearContents2" value="true"/>
<booleanAttribute key="org.eclipse.m2m.qvt.oml.interpreter.clearContents3" value="true"/>
<mapAttribute key="org.eclipse.m2m.qvt.oml.interpreter.configurationProperties"/>
<intAttribute key="org.eclipse.m2m.qvt.oml.interpreter.elemCount" value="3"/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.featureName1" value=""/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.featureName2" value=""/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.featureName3" value=""/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.module"
 value="platform:/resource/QVTTransformations/add_component.qvto"/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.targetModel1"
 value="platform:/resource/QVTTransformations/cxf_transport_2.6.archabst"/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.targetModel2"
 value="platform:/resource/QVTTransformations/addUDPComponentToCXF.archabst"/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.targetModel3"
 value="platform:/resource/QVTTransformations/transformed/cxf_transport_2.6.withUDPComponent.archabst"/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.targetType1" value="NEW_MODEL"/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.targetType2" value="NEW_MODEL"/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.targetType3" value="NEW_MODEL"/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.traceFile"
 value="platform:/resource/QVTTransformations/out2.archabst.qvtotrace"/>
<booleanAttribute key="org.eclipse.m2m.qvt.oml.interpreter.useTraceFile" value="true"/>
</launchConfiguration>
```

Figure B.26: Exemplary launch configuration for adding the new UDP component to CXF architecture abstraction specification.

References

[1] A. Ahmad, P. Jamshidi, and C. Pahl. Classification and comparison of architecture evolution reuse knowledge-a systematic review. *J. Softw. Evol. Process*, 26(7):654–691, July 2014.

[2] O. Barais, A. F. Le Meur, L. Duchien, and J. Lawall. Software Architecture Evolution. In T. Mens and S. Demeyer, editors, *Softw. Evol.*, pages 233–262. Springer Berlin Heidelberg, 2008.

[3] J. M. Barnes. NASA's advanced multimission operations system. In *8th Int. ACM SIGSOFT Conf. Qual. Softw. Archit. (QoSA '12)*, pages 3–12, New York, New York, USA, 2012. ACM Press.

[4] J. M. Barnes, D. Garlan, and B. R. Schmerl. Evolution styles: foundations and models for software architecture evolution. *Softw. Syst. Model.*, 13(2):649–678, Nov. 2012.

[5] J. M. Barnes, A. Pandey, and D. Garlan. Automated planning for software architecture evolution. In *28th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE 2013)*, pages 213–223. IEEE, Nov. 2013.

[6] E. H. Bersoff, V. D. Henderson, and S. G. Siegel. *Software configuration management. An investment in product integrity.* Addison-Wesley, New York, New York, USA, 1980.

[7] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby. COCOMO 2.0. *Ann. Softw. Eng.*, 1(1):1–24, 1995.

[8] J. Bosch. Software Architecture: The Next Step. In *1st Eur. Work. Softw. Archit.*, pages 194–199, Heidelberg, 2004. Springer.

[9] L. Bratthall, E. Johansson, and B. Regnell. Is a Design Rationale Vital when Predicting Change Impact? A Controlled Experiment on Software. In *2nd Int. Conf. Prod. Focus. Softw. Process Improv. (PROFES 2000)*, pages 126–139. Springer, 2000.

[10] H. P. Breivold, I. Crnkovic, and M. Larsson. A systematic review of software architecture evolution research. *Inf. Softw. Technol.*, 54(1):16–40, Jan. 2012.

[11] J. Brunet, R. A. Bittencourt, D. Serey, and J. Figueiredo. On the Evolutionary Nature of Architectural Violations. In *19th Work. Conf. Reverse Eng.*, pages 257–266, Kingston, Oct. 2012. IEEE Computer Society.

[12] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies.* Addison-Wesley Professional, 2001.

[13] A. Corazza, S. Di Martino, and G. Scanniello. A probabilistic based approach towards software system clustering. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 88–96, Washington, DC, USA, 2010. IEEE Computer Society.

[14] R. Correia, C. Matos, M. El-Ramly, R. Heckel, G. Koutsoukus, and L. Andrade. Software Engineering at the Architectural Level: Transformation of Legacy Systems. Technical report, University of Leicester, 2002.

[15] C. E. Cuesta, E. Navarro, D. E. Perry, and C. Roda. Evolution styles: using architectural knowledge as an evolution driver. *J. Softw. Evol. Process*, 25(9):957–980, Sept. 2013.

[16] R. C. de Boer, P. Lago, A. Telea, and H. van Vliet. Ontology-driven visualization of architectural design decisions. In *Jt. Work. IEEE/IFIP Conf. Softw. Archit. Eur. Conf. Softw. Archit. (WICSA/ECSA 2009)*, pages 51–60. IEEE, Sept. 2009.

[17] W. Ding, P. Liang, A. Tang, and H. van Vliet. Knowledge-based approaches in software documentation: A systematic literature review. *Inf. Softw. Technol.*, 56(6):545–567, June 2014.

[18] A. Egyed. Consistent Adaptation and Evolution of Class Diagrams during Refinement. In *Fundam. Approaches to Softw. Eng. (FASE 2004)*, pages 37–53, Barcelona, Spain, 2004. Springer Berlin Heidelberg.

[19] M. Feilkas, D. Ratiu, and E. Jurgens. The loss of architectural knowledge during system evolution: An industrial case study. In *17th IEEE Int. Conf. Progr. Compr.*, pages 188–197. IEEE Computer Society Press, May 2009.

[20] D. Ganesan and M. Lindvall. Adam: External dependency-driven architecture discovery and analysis of quality attributes. *ACM Trans. Softw. Eng. Methodol.*, 23(2):17:1–17:51, Apr. 2014.

[21] T. Haitzer, E. Navarro, and U. Zdun. Architecting for decision making about code evolution. In *1st International Workshop on Software Architecture Asset Decision-Making (SAADM) co-located with the 9th European Conference on Software Architecture (ECSA 2015)*, September 2015.

[22] T. Haitzer and U. Zdun. Controlled Experiment on the Supportive Effect of Architectural Component Diagrams for Design Understanding of Novice Architects. In *7th Eur. Conf. Softw. Archit. (ECSA 2013)*, volume 7957, pages 54–71, 2013.

[23] T. Haitzer and U. Zdun. Semi-automated architectural abstraction specifications for supporting software evolution. *Sci. Comput. Program.*, 90:135–160, Sept. 2014.

[24] N. B. Harrison, P. Avgeriou, and U. Zdun. Using Patterns to Capture Architectural Decisions. *IEEE Softw.*, 24(4):38–45, 2007.

[25] I. Herraiz, D. Rodriguez, G. Robles, and J. M. Gonzalez-Barahona. The evolution of the laws of software evolution. *ACM Comput. Surv.*, 46(2):1–28, Nov. 2013.

[26] D. Heuzeroth, T. Holl, G. Hogstrom, W. Lowe, G. Högström, and W. Löwe. Automatic design pattern detection. In *2003 Int. Symp. Micromechatronics Hum. Sci.*, pages 94–103, 2003.

[27] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design Science in Information Systems Research. *MIS Q.*, 28(1):75–105, 2004.

[28] R. Holt. Sofware Architecture as a Shared Mental Model. In University of Alberta, editor, *Proc. ASERC Work. Softw. Archit.*, 2002.

[29] S. Hunold, M. Korch, B. Krellner, T. Rauber, T. Reichel, and G. Rünger. Transformation of Legacy Software into Client/Server Applications through Pattern-Based Rearchitecturing. In *32nd Annu. IEEE Int. Comput. Softw. Appl. Conf.*, pages 303–310. IEEE, 2008.

[30] ISO/IEC/IEEE. Systems and software engineering – architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pages 1 –46, 1 2011.
[31] P. Jamshidi, M. Ghafari, A. Ahmad, and C. Pahl. A Framework for Classifying and Comparing Architecture-centric Software Evolution Research. In *2013 17th Eur. Conf. Softw. Maint. Reengineering*, pages 305–314. IEEE, Mar. 2013.
[32] M. A. Javed, S. Stevanetic, and U. Zdun. Cost-effective traceability links for architecture-level software understanding: A controlled experiment. In *Proceedings of the 24th Australasian Software Engineering Conference*, ASWEC. ACM, 2015.
[33] M. A. Javed and U. Zdun. The supportive effect of traceability links in architecture-level software understanding: Two controlled experiments. In *Proceedings of the 11th Working IEEE/IFIP Conference on Software Architecture*, WICSA 2014, pages 215–224. IEEE, 2014.
[34] M. A. Javed and U. Zdun. On the effects of traceability links in differently sized software systems. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, EASE 2015. ACM, 2015.
[35] M. A. Javed and U. Zdun. The supportive effect of traceability links in change impact analysis for evolving architectures – two controlled experiments. In *14th International Conference on Software Reuse*, ICSR 2015. Springer link, 2015.
[36] C. Jones and O. Bonsignour. *The economics of software quality*. Addison-Wesley Professional, 2011.
[37] J. Knodel, M. Lindvall, D. Muthig, and M. Naab. Static evaluation of software architectures. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 10 pp.–294, March 2006.
[38] J. Knodel, D. Muthig, and D. Rost. Constructive architecture compliance checking - an experiment on support by live feedback. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 287–296, Sept 2008.
[39] C. Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *4rd Work. Conf. Reverse Eng. (WCRE '96)*, pages 208–215. IEEE Comput. Soc. Press, 1996.
[40] O. Le Goaer, D. Tamzalit, M. C. Oussalah, and A.-D. Seriai. Evolution styles to the rescue of architectural evolution knowledge. In *3rd Int. Work. Shar. Reusing Archit. Knowl.*, pages 31–36, New York, New York, USA, 2008. ACM Press.
[41] M. Lehman. Programs, life cycles, and laws of software evolution. *Proc. IEEE*, 68(9):1060–1076, 1980.
[42] M. M. Lehman. Laws of software evolution revisited. In *5th Eur. Work. Softw. Process Technol.*, pages 108–124, Nancy, 1996. Springer Berlin Heidelberg.
[43] Z. Li, P. Liang, and P. Avgeriou. Application of knowledge-based approaches in software architecture: A systematic mapping study. *Inf. Softw. Technol.*, 55(5):777–794, May 2013.
[44] M. Lindvall, W. Stratton, D. Sibol, C. Ackermann, W. Reid, D. Ganesan, D. McComas, M. Bartholomew, and S. Godfrey. Connecting research and practice: an experience report on research infusion with software architecture visualization and evaluation. *Innovations in Systems and Software Engineering*, 8(4):255–277, 2012.
[45] I. Lytra, H. Tran, and U. Zdun. Supporting Consistency between Architectural Design Decisions and Component Models through Reusable Architectural Knowledge Transformations. In *7th Eur. Conf. Softw. Archit. (ECSA 2013)*, pages 224–239. Springer Berlin Heidelberg, 2013.
[46] H. J. Macho and G. Robles. Preliminary lessons from a software evolution analysis of Moodle. In *First Int. Conf. Technol. Ecosyst. Enhancing Multicult. - TEEM '13*, pages 157–161, New York, New York, USA, 2013. ACM Press.
[47] K. Mens, T. Mens, and M. Wermelinger. Maintaining software through intentional source-code views. In *14th Int. Conf. Softw. Eng. Knowl. Eng. (SEKE '02)*, pages 289–296, New York, July 2002. ACM Press.
[48] Moodle. Moodle. `https://moodle.com`. Last access July 20th, 2016.
[49] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between design and implementation. *IEEE Trans. Softw. Eng.*, 27(4):364–380, 2001.
[50] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '95, pages 18–28, New York, NY, USA, 1995. ACM.
[51] E. Y. Nakagawa, E. P. Machado de Sousa, K. de Brito Murata, G. de Faria Andery, L. B. Morelli, and J. C. Maldonado. Software Architecture Relevance in Open Source Software Evolution: A Case Study. *32nd Annu. IEEE Int. Comput. Softw. Appl. Conf. (COMPSAC 2008)*, pages 1234–1239, 2008.
[52] E. Navarro and C. E. Cuesta. Automating the Trace of Architectural Design Decisions and Rationales Using a MDD Approach. In *Second Eur. Conf. Softw. Archit. (ECSA 2008)*, pages 114–130. Springer, 2008.
[53] E. Navarro, C. E. Cuesta, D. E. Perry, and P. González. Antipatterns for Architectural Knowledge Management. *Int. J. Inf. Technol. Decis. Mak.*, 12(3):547–589, 2013.
[54] I. Neamtiu, G. Xie, and J. Chen. Towards a better understanding of software evolution: an empirical study on open-source software. *J. Softw. Evol. Process*, 25(3):193–218, Mar. 2013.
[55] J. Noppen and D. Tamzalit. ETAK: Tailoring Architectural Evolution by (re-)using Architectural Knowledge. In *ICSE Work. Shar. Reusing Archit. Knowl. (SHARK '10)*, pages 21–28, New York, New York, USA, 2010. ACM Press.
[56] I. Ozkaya, R. Kazman, and M. Klein. Quality-attribute based economic valuation of architectural patterns. In *First Int. Work. Econ. Softw. Comput. ESC'07*, 2007.
[57] I. Ozkaya, P. Wallin, and J. Axelsson. Architecture knowledge management during system evolution. In *2010 ICSE Work. Shar. Reusing Archit. Knowl. (SHARK '10)*, pages 52–59, Helsinki, May 2010. ACM Press.
[58] C. Pahl, S. Giesecke, and W. Hasselbring. Ontology-based modelling of architectural styles. *Inf. Softw. Technol.*, 51(12):1739–1749, Dec. 2009.
[59] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *ACM Softw. Eng. Notes*, 17(4):40–52, 1992.
[60] G. Phipps. Comparing observed bug and productivity rates for java and c++. *Softw., Pract. Exper.*, 29(4):345–358, 1999.
[61] J. Rosik, A. Le Gear, J. Buckley, M. A. Babar, and D. Connolly. Assessing architectural drift in commercial software development: a case study. *Softw. Pract. Exp.*, 41(1):63–86, Jan. 2011.

[62] K. Sartipi. Software architecture recovery based on pattern matching. In *19th Int. Conf. Softw. Maint. (ICSM 2003)*, pages 293–296. IEEE Comput. Soc, 2003.
[63] J. Sillito, G. C. Murphy, and K. De Volder. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering*, 34(4):434–451, jul 2008.
[64] C. Stoermer, A. Rowe, L. O'Brien, and C. Verhoef. Model-centric software architecture reconstruction. *Softw. Pract. Exp.*, 36(4):333–363, Apr. 2006.
[65] D. Tamzalit, M. C. Oussalah, O. Le Goaer, and A.-D. Seriai. Updating software architectures : A style-based approach. In *Int. Conf. Softw. Eng. Res. Pract. (SERP 2006)*, pages 313–318, Las Vegas, 2006. CSREA Press.
[66] A. Tang, A. B. Muhammad, I. Gorton, and J. Han. A survey of architecture design rationale. *J. Syst. Softw.*, 79(12):1792–1804, Dec. 2006.
[67] A. Tang, A. E. Nicholson, Y. Jin, and J. Han. Using Bayesian belief networks for change impact analysis in architecture design. *J. Syst. Softw.*, 80(1):127–148, Jan. 2007.
[68] P. Tarvainen. Adaptability Evaluation of Software Architectures: A Case Study. In *31st Annu. Int. Comput. Softw. Appl. Conf. - Vol. 2 - (COMPSAC 2007)*, pages 579–586. IEEE, July 2007.
[69] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha. A recommendation system for repairing violations detected by static architecture conformance checking. *Softw. Pract. Exp.*, pages n/a–n/a, Sept. 2013.
[70] C. Tibermacine, R. Fleurquin, and S. Sadou. A family of languages for architecture constraint specification. *J. Syst. Softw.*, 83(5):815–831, May 2010.
[71] D. Tofan, M. Galster, P. Avgeriou, and W. Schuitema. Past and Future of Software Architectural Decisions a Systematic Mapping Study. *Inf. Softw. Technol.*, Mar. 2014.
[72] Tracker Moodle. Tracker Moodle. `https://tracker.moodle.org/secure/Dashboard.jspa`. Last access July 20th, 2016.
[73] V. K. Vaishnavi and W. Kuechler. *Design Science Research Methods and Patterns: Innovating Information and Communication Technology*. Auerbach, 2007.
[74] B. S. von Detten M., M. von Detten, and S. Becker. Combining clustering and pattern detection for the reengineering of component-based software systems. In *2011 Fed. Events Component-Based Softw. Eng. Softw. Archit.*, pages 23–32, 2011.
[75] R. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, 2014.