# Towards Architecture-Centric Evolution of Long-Living Systems (The ADVERT Approach)

Marco Konersmann*, Zoya Durdik**, Michael Goedicke*, Ralf H. Reussner**

*{marco.konersmann,michael.goedicke}@paluno.uni-due.de
paluno - The Ruhr Institute for Software Technology
University of Duisburg-Essen
Essen, Germany

**{zoya.durdik,reussner}@kit.edu
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

## ABSTRACT

Although an intensive research attention has been paid to software evolution, there is no established approach which supports a software development and evolution round-trip between requirements, design decisions, architectural elements, and code. The ADVERT approach shall provide support for software evolution on an architectural level. ADVERT is based on two core ideas: (1) Maintaining trace links between requirements, design decisions, and architecture elements, and (2) explicitly integrating software architecture information into the code. The expected benefits of the approach are: (1) Eased understanding of the relationship between requirements and design, and (2) assured compliance between architectural design and implementation. In this position paper we explain our envisioned approach and demonstrate it on a CoCoME-based example, which is a benchmark for component-based modelling approaches.

## Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

## Keywords

Software architecture, Software evolution, Design decisions, Embedded architecture

## 1. INTRODUCTION

Software systems are naturally change-prone and evolve over time. The changes are caused by internal factors, such as bugs or performance profiling, and by external factors, such as new requirements or changing technology. Implementation of changes typically affects multiple artefacts, such as system specifications, architectural models, and code. However, the changes are often unsystematic and bypass one or several of the artefacts, in particular system documentation and architectural models. Thus, the artefacts become out of synch, system design drifts over time and system evolution becomes expensive and time-consuming. Especially long-living systems (systems with the life span over 10 years) are affected by these problems due to evolution.

What are the reasons for these problems? One of the reasons is that software evolution is neglected during the system design, as we have discovered in our previous work [3]. The neglecting is caused by time and budget pressure, and also by the fact that the consequences for the evolution are actually not always clear to the developers.

Another reason is the system complexity. Designing software architecture includes numerous design decisions triggered by requirements as well as technical and management constraints. These decisions are often taken by different persons and under various conditions. Therefore, documentation of decisions together with the rationale is of special importance for system evolution. However, documentation of decisions with rationale and trace links to the requirements is a manual process. It is cumbersome, error-prone and time-consuming.

A third reason are the at least two representations of architecture: the architecture specification and the implementation, whereby some parts of the architecture information are only visible in one of the representations. These two representations have to be synchronised to avoid misinformation. Often the documentation and architectural models are out of date or are simply missing, and the code is the only remaining documentation of the system design.

A significant research effort has been directed to the problem of software evolution [16], however, we could not discover any mature industry-applied approach to support the development and evolution of long-living systems [3].

In this position paper we explain our envisioned architecture–centric approach ADVERT to support system development and evolution round-trip between requirements, design decisions, architectural elements, and code. We demonstrate the approach on a CoCoME-based example, which is a benchmark for component-based modelling approaches [10]. The ADVERT approach shall provide support to evolution of long-living systems on an architectural level. It is based on two core ideas: The maintenance of trace links between requirements and architectural elements through design decisions, and the integration of software architecture information with the code. The expected benefits of the approach are: (1) To explicitly show the consequences of design decisions for the evolvability of the system by annotating the solutions with information for the evaluation of their appli-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

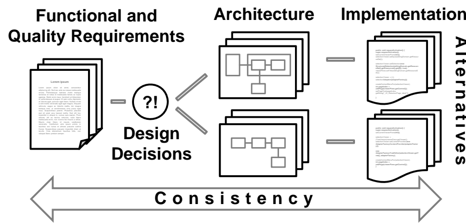



**Figure 1: Approach overview**

cability, (2) to make complex systems easier to manage by giving support in documenting design decisions and generating and maintaining trace links, and (3) to avoid misinformation from outdated architecture artefacts by eliminating redundant architecture representations.

In section 2 we explain the envisioned approach and provide an example of its use. In section 3 we discuss related work. Section 4 concludes the paper and draws future work.

# 2. ARCHITECTURE-CENTRIC EVOLUTION OF LONG-LIVING SYSTEMS

In this position paper we present an approach for architecture–centric evolution of long–living systems. The goal of this approach is to support the evolution of systems by tightly integrating requirements engineering, architecture design, and implementation. To achieve this goal, we support the consistency between functional and quality requirements, design decisions, architecture specification, and architecture implementation, and provide support to systematically evaluate architecture alternatives (cf. figure 1).

Within this approach we extend and seamlessly combine two methods that we have formerly presented separately. The combination is shown in figure 2. Part (a) of figure 2 outlines the core concepts of the first method for documenting design decisions and maintaining trace links between requirements, design decisions and architecture. This method is the extension of [4]. It uses existing architecture specifications in an architecture description language (ADL). The results of this method are changes to these architecture specifications and decisions documented together with rationale.

Part (b) of figure 2 outlines the core concepts of the second method. This is a method for explicitly integrating architecture with program code, and eliminating redundant architecture representations [12]. The second method uses architecture specifications in an ADL, and integrates this information explicitly into program code. The result of this method is executable source code that implies the architecture specification. When the architecture is changed in the code, these changes are reflected in the architecture specification. Therefore the architecture specification in an ADL is the interface that easily allows to combine the methods to form an architecture-centric approach to evolution in long-living systems (ADVERT).

In the following, we present the use of the envisioned combined approach within an exemplary evolution scenario. We first outline the scenario. Then we explain the combined methods, before we demonstrate the application of the combined methods in the exemplary scenario.

## 2.1 Exemplary Evolution Scenario

The evolution scenario is divided into seven steps: In step (1) change requests arrive that are relevant to the architecture. Prior requirements are invalidated or new requirements are added to the system due to these changes. In step (2) prior decisions affected by the requirements change are re-evaluated or new design decisions are taken. These decisions can be identified via predefined trace links from the requirements. In this step affected or new architectural elements are identified by following trace links from changed or new design decisions. In step (3) the architecture (expressed in an ADL) is adapted with respect to the changed and new design decisions. In step (4) the changes in the architecture are automatically transferred to the code via an intermediate architecture language. In this process architecture information is explicitly integrated with the program code so that it can be extracted easily. In step (5) the code is completed by programmers to reflect the detailed behaviour of the software on a non-architectural level. In step (6) the architecture description is extracted from the code via the formerly mentioned intermediate language. In step (7) the architecture is compared to the intended architecture and validated by architects.

To use our approach in this evolution scenario, it is required that the system has formerly been developed using that approach. The initial development of the system is to be made in a scenario like above. Such a scenario would, however, only create new requirements in the first step.

In the following sections we provide an explanation of the ADVERT approach. The explanation is divided into two parts. The first part deals with the steps 1 to 3. It covers the evolution from change requests to revised design decisions and their effect on the architecture specification. The second part deals with the steps 4 to 7 and covers the explicit integration of the changed architecture specification with program code. In the last section we demonstrate our approach on a real-life-based example following the scenario outlined above.

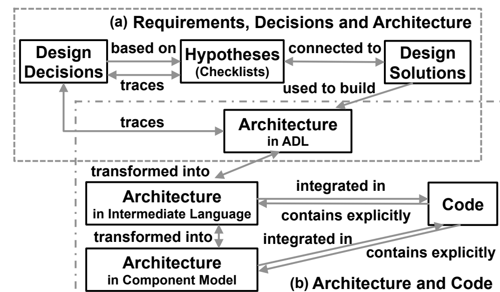


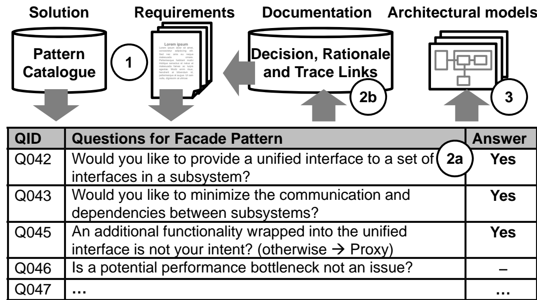

**Figure 2: (a) outlines the core concepts for documenting design decisions and maintaining trace links between requirements, design decisions, and architecture artefacts. (b) outlines the core concepts for explicitly integrating architecture with program code, and eliminating redundant architecture. The architecture in an ADL is the combining element.**

## 2.2 Linked decisions and architecture

The core idea of this part of the ADVERT approach is to distinguish between two types of design solutions, custom and recurring design solutions, such as design patterns, components and web services. The idea is based on our earlier proposed catalogue of design patterns [4, 5], which we extend

for ADVERT to support web services and components. The descriptions of recurring design solutions in the catalogue are stored together with solution-specific questions. These questions serve as checklists to validate the applicability of the selected solution in the problem context. An example of questions to a catalogue solution "Facade pattern" is provided on figure 3. Answered questions are rationale for the decision to select or to withdraw the solution. Please refer to our previous work for more information on catalogue questions and answers to them [4, 5].

Once change requests arrive, engineers select a potential solution (step 1 on figure 3) to satisfy new or changes requirements using a state-of-the-art approach for a solution selection (see section 3). If the selected solution is a reusable solution, software engineers answer the questions from the catalogue to evaluate the applicability of the solution for the specific problem (step 2a on figure 3). Answers to the questions from the catalogue are automatically saved as a rationale together with decisions to take or to withdraw the solution (step 2b on figure 3). Based on the rationale, potentially affected design decisions can be easier verified by software engineers for their validity. The new or modified requirements are automatically linked to the decision as its triggers.




| QID | Questions for Facade Pattern | Answer |
|---|---|---|
| Q042 | Would you like to provide a unified interface to a set of interfaces in a subsystem? | Yes |
| Q043 | Would you like to minimize the communication and dependencies between subsystems? | Yes |
| Q045 | An additional functionality wrapped into the unified interface is not your intent? (otherwise → Proxy) | Yes |
| Q046 | Is a potential performance bottleneck not an issue? | – |
| Q047 | ... | ... |

**Figure 3: An example catalogue entry: Facade pattern questions with answers**

If a new decision is met based on new or changed requirements or an invalid decision is encountered, an engineer updates the architecture specification (step 3 on figure 3). The decision is semi-automatically linked to the new or changes architectural elements. Thus, new or updated decisions are saved together with links to architectural elements and requirements and a rational (in form of answers to the questions) provided by an engineer. The architecture specification is the input for the next step of the approach.

As custom design solutions can not be anticipated, they can not be captured and provided with checklists in a catalogue in advance. Thus, we cannot ease the documentation of such decisions with the solution catalogue. For these cases other approaches have to be used, such as problem frames [11] to come to a solution, and trace meta-models [6] to link artefacts. While the creation of trace links and capture of the decision rationale can not be fully automated, we hope to lower the burden of documentation with the help of pre-annotated design solutions.

The results of this part of the ADVERT approach are: Evaluated and semi-automatically documented design decisions, rationale for the decisions, and trace links connecting design decisions to the triggering requirements and to the implementing architectural elements.

## 2.3 Explicitly Integrated Architecture

In software architecture, usually at least two representations of the architecture exist: The architecture is explicitly documented in a specification and mostly implicitly encoded in the implementation. A specification is expressed in an ADL such as UML. In business information systems the architecture implementation often follows the structure of component models (CM). Both representations of the architecture, expressed in ADLs and CMs, are necessary to fully represent the system architecture. These two representations have to be synchronised to avoid misinformation, because they share information. The goal of the Explicitly Integrated Architecture approach (presented formerly in [12]) is to avoid differences between these representations. To achieve this goal, the specification is explicitly integrated into the program code in such a way that it complies a CM, and at the same time includes the additional information from the architecture specification (e.g. cf. [15]). Therefore only one representation remains.

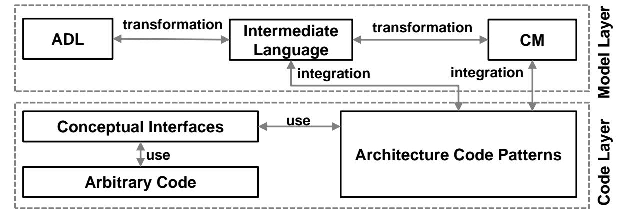



**Figure 4: Overview of the Explicitly Integrated Architecture Approach**

Figure 4 shows the concept of Explicitly Integrated Architecture. The approach includes two steps: In the step (a) the architecture is modelled using an ADL and transformed to match the CM of choice. This introduces to two challenges: (1) There are several ADLs and several CMs. This is a n:m relationship which does not scale well. (2) CMs and ADLs differ in their features and their abstraction level. CMs typically only define the structure of the system formally, and are highly integrated with the implementation details. ADLs hide implementation details, but typically include aspects besides the structure, e.g. behaviour or quality (cf. [14]). ADLs among each other also differ in their features, as do CMs.

To address (1) we introduce an intermediate language (IL) that reduces the n:m relationship to a n:1:m relationship. We create bidirectional transformations between ADLs and the IL, and between the IL and the CMs (e.g. see [12]). To address (2), the IL and their relationship to ADLs and CMs have to respect the variability of these languages. Features of ADLs and CMs that have a direct equivalent in another representation can be used in both representations by defining a model transformation. When no equivalent exists, a complex transformation should be defined that emulates the missing feature, if possible. If the feature cannot be emulated, it cannot be used in the first place or has to be integrated as uninterpreted IL elements. If the feature is mandatory to be used (e.g. because it is a core concept of the language), but it cannot be expressed in the second representation, the representations are incompatible. Thus, choosing the first representation excludes the latter from being chosen. To consider this aspect, the meta model of the intermediate language is modular (cf. [12]).

There is information in ADLs that cannot be expressed in certain CMs, and can also not be emulated by complex transformations as described earlier. This is typically behaviour and quality, but arbitrary other information is imaginable. Behaviour information can be integrated as integrated behavioural model, as has already been shown with process models and automata [1]. Quality and other information may need to be integrated as uninterpreted data.

The step (b) is to create source code that is structured so that it represents the model elements as denoted in the CM view. This is what we call integrated CM. We want to create a bidirectional mapping between code patterns[1] and CM elements. A simple example is an Enterprise JavaBeans (EJB – a widely used component framework for enterprise systems) singleton bean, which maps to a Java class with the annotation `@Singleton`. The name of the bean is the name of the class. The interpretation of this information at run time is subject to the execution environment. Other mappings, especially for behavioural specifications can be more complex.

Our approach does not aim to replace the complete implementation phase, but to replace the coding of architecturally relevant code. The detailed behaviour of atomic components has still to be implemented by hand or other approaches. The code patterns generated by our approach use conceptual interfaces to interact with such arbitrary other code. These are not necessarily technical interfaces. In the example of the singleton bean in EJB the conceptual interface for arbitrary code that defines the bean's behaviour is the location of the code within the class that represents the bean.

In step (a) bidirectional transformations were defined between ADLs and CMs via an IL. In step (b) the information of the CM has been explicitly integrated into the program code, so that it can be unambiguously extracted from the code. Following this approach, a software architecture implementation is generated that corresponds to the architecture modelled in an ADL. Also the code can be parsed and the underlying ADL model can be shown. Therefore this part implements the steps (4) and (6) of our evolution scenario in section 2.1.

Using this approach allows to extract the actual architecture from the code. It requires discipline from the developers to not break the integrated architecture. Arbitrary architectural changes can be made in the source code. When the changes follow the defined code patterns, such changes can be viewed when extracting the architecture. However developers can always break the architecture invisible to our approach, e.g. by using communication channels not considered by the integration approach.

When an integrated architecture is changed in an evolutionary manner, the underlying code base is adapted to reflect the changes. The conceptual interfaces have to be respected during the adaptation, for the other code to work properly. It cannot be guaranteed that the code describing the detailed behaviour is still working as expected. Therefore tests will be necessary.

## 2.4 Demonstration on example

We demonstrate our approach with the exemplary scenario shown in section 2.1 using Common Component Modeling Example (CoCoME) [10]. CoCoME is an example system with the original goal to define a benchmark for component models. It consists of cash desks with terminals to buy items, store servers to manage the stock of a single store, enterprise servers to manage the stock of several stores, and the respective clients to use the servers. Imagine that the Hexxon petrol station group has a CoCoME installation to sell and manage items in their petrol stations. Hexxon CoCoME was designed and documented using the ADVERT approach. Design decisions are documented together with rationale and trace links to requirements and architectural elements. There is an explicitly integrated architecture available. For this scenario we have adopted the architecture of the CoCoME enterprise server, which is provided on figure 5 together with a high-level view of CoCoME.

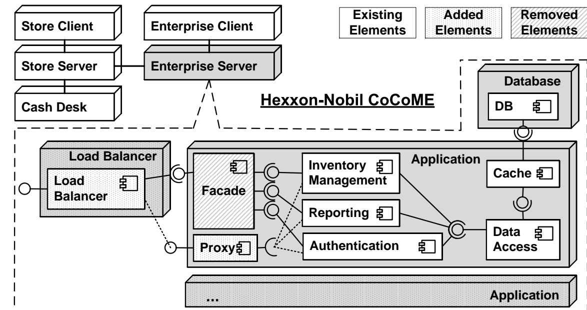



**Figure 5: Hexxon-Nobil CoCoMe architecture with required changes**

In our scenario, the Hexxon and Nobil petrol station groups merge. Hexxon-Nobil wants to use CoCoME installations in all of their stores.

**Linked decisions and architecture**

In **step (1)** of our evolution scenario, the following changes are requested: (C1) Due to the expanded installation the system needs to support 1400 petrol stations instead of 700. The enterprise servers cannot handle this load, as identified by a performance analysis. (C2) As the system has to be changed, the company also plans to upgrade the underlying technological platform from Enterprise Java Beans (EJB) 2.1 to EJB 3.1.

In **step (2)**, software engineers identify the possible solutions to meet the changes: C1 involves (see figure 5): new component "Load Balancer", replication and deployment on additional hardware knots, and evtl. reconfiguration of Hibernate. C2 involves several code changes for the newer EJB standard uses a new programming style than the old standard. The architecture itself remains the same.

C1 involved implementation of a new component "Load Balancer". This component shall be connected before the "Facade" component and deployed on a dedicated server. As a "Facade" component might be a potential bottleneck, software engineers want to re-evaluate the decision to use a Facade. They check the rationale saved for the decision — answers to the questions from the solution catalogue (figure 3). They discover question number Q045 "An additional functionality wrapped into the unified interface is not your intent? (otherwise use Proxy)" answered as "Yes". This is a contradiction, as the application components will be replicated and the Facade component would need to manage the sessions. Software engineers refer to the solution catalogue

[1]We do not mean patters in terms of the GoF [7], but templates of structural elements and instructions within the source code. E.g. a class with an annotation that indicates a state definition.

to evaluate the recommended Proxy pattern and answer the provided questions. In particular, the question Q035 "Would you like to provide an interface with some additional functionality, e.g. management of objects, etc.?" is conform with the new requirements. A decision to replace Facade pattern component with the Proxy pattern component is taken.

Thus, C1 involves not only new component "Load Balancer", replication and deployment on additional hardware knots, and evtl. reconfiguration of Hibernate, but also replacement of Facade component with the Proxy one (see figure 5) . In **step (3)** engineers update the architectural specification in the ADL. They revise requirements connected to the Facade decision. The requirements and the involved architectural elements are discovered with the help of the saved trace links. In the background, the new links between C1, changes to requirements, decisions and architectural elements are captured.

There is a second way to discover that the Facade design decision is deprecated. Software engineers could have started analysing the requirements to the system. They would have discovered the requirement R023, which was one of the contributing requirements to the Facade decision. This requirement is outdated due to the C1. Software engineers are suggested a list of design decisions, where the deprecated requirement triggered the decision. The decision to use the Facade pattern is re-evaluated as described above

**Explicitly Integrated Architecture**

The system architecture is implemented using EJB 2.1. Here we assume that the code has already been developed using our approach. Therefore the actual architecture can be extracted and viewed in an UML architectural representation. The ADL view used in step (3) was already an extracted, integrated architecture. The source code was parsed and the EJB 2.1 CM representation was extracted. This representation was transformed into a model of the IL meta model, which was in turn transformed into a model conforming the UML meta model.

In **step (4)** the ADL representation is integrated into the code. C1 includes a new component "Load Balancer", which is modelled as a basic component in UML. One single instance is allocated to a deployment container. The transformation into the EJB 2.1 results in a new singleton bean. Architectural code for this bean can be generated, including an interface and method skeletons. C1 also implies a replacement of the "Facade" with a "Proxy". The old facade bean is deleted while a new proxy bean is created with new connections. These changes are made analogously to the new Load Balancer component.

In **step (5)** the detailed behaviour of the load balancer needs to be implemented by a coder in this example.

The replication modelled in UML for C1 results in the deployment of copies of the application package on multiple nodes. There is no deployment information in the EJB 2.1 component model. The information about the replication can be attached to the replicated component using an annotation, so that this information is available when the architecture is extracted. But the replicated deployment has to be done manually, therefore this is also executed in step (5). Deployment information could e.g. be managed by a deployment infrastructure, but the EJB specification does not specify such an infrastructure.

C2 implies a change from the EJB 2.1 CM to EJB 3.1. The newer specification includes a new programming style, relying more on annotations for defining components and connections than EJB 2.1, which is configured mainly using XML descriptor files. For changing from the old to the new programming model, the meta models for EJB 2.1 and EJB 3.1 are necessary, as well as transformations between the IL meta model and both EJB versions. The architecture needs to be extracted from the old code, and the transformation target needs to be set to EJB 3.1. The architecture is the completely changed to EJB 3.1. The non-architectural code might not work as expected, where it relies on the old specification, but when the conceptual interfaces between the architectural code patterns and arbitrary other code is respected, no more work is necessary.

In **step (6)** of the scenario, the new actual architecture is extracted from the code. The information is transformed into CM and IL representations, which are in turn transformed into an ADL representation. In **step (7)** the architecture is reviewed by an architect. In this demonstration the manual changes did not change the architecture and can therefore be confirmed.

# 3. RELATED WORK

We have investigated the evolution of long-living systems in our previous work [16, 3], and have proposed guidelines to be considered during systems' life-cycles to reduce the evolution problems [3]. These guidelines contain well-approved software engineering methods and tools, and are not a uniform approach to accompany systems' design and evolution.

In this position paper we propose a uniform approach to tightly integrate architecture design, implementation, and requirements to ease evolution of long-living systems. The approach extends and combines our previous work [4, 12] and consists of (1) an architecture language supporting evolution, (2) a catalogue of reusable solutions with question annotations to evaluate the solutions, (3) a method to documents decisions together with rationale and to establish trace links between artefacts, and (4) explicitly integrated architecture. We structure the related work according to these four aspects. Due to space reasons we provide only a few related work approaches per aspect.

ADLs have been subject to research for decades, but they are often focused on specific domains, are typically not interchangeable, and do not consider longevity and future evolution [3]. Until now we found no ADL that explicitly supports evolution. In [9] a tool building upon an ADL is presented that introduces evolution paths as first class entities. We plan to support such concepts with our approach.

The idea to structure the information in catalogues is not new. Some examples are books such as Gamma et al. [7] and other catalogues e.g. by Tichy [19]. In addition, some approaches also support the selection of the solutions from the catalogue. Wang et al. [20] propose an approach to guide the selection of solutions based on quality properties. Garbe et al. [8] propose KARaCAs approach, which is an expert system based on the Bayesian Belief Network and where questions are used to select the most appropriate pattern. These approaches are complimentary to our approach. The unique feature of our approach is the inclusion of question annotations to evaluate the solutions and to semi-automatically capture the rationale for decisions to select a solution.

In order to capture design decisions and rationale metamodels have been proposed e.g., by Kruchten [13] or Tang et al. [18]. A survey by Galvao et al. [6] discusses and evaluates

the state-of-the-art in traceability approaches according to the traceability representation, mapping, scalability, change impact analysis, and tool support. Although traceability is widely researched in academia [6], the results the have not yet reached the maturity to be applied in industrial context [2]. Furthermore, additional research is needed to develop methods and tool support for rationale capturing and usage, as to Tang et al. [17]. The major problem here is the high overhead, which can be reduced with the pre-annotated reusable solutions proposed in our approach.

Explicitly Integrated Architecture is based on the idea in [1]. However, Balz considers mainly behavioural models. Architecture models contain multiple view points that must be synchronised. Balz arguments in [1] why no directly competing work exists.

# 4. CONCLUSION AND FUTURE WORK

In this paper we presented our approach for software evolution on an architectural level. We provide trace links between requirements, design decisions, and architecture elements. Our architecture is embedded into the source code. The trace links and the integration combined provide the means to achieve the goal of the overall approach: supporting the evolution of long-living systems, by making the decision process transparent and the architecture manageable.

Some parts of the approach described in this paper have already been implemented, e.g. meta models for the design decision process and the traces have been developed, and an initial prototype for model transformations between ADLs and Component Models exists. In short term future work we will extend these activities to fully implement the approach. In future work we plan to develop an integrated tool chain that consistently supports the overall approach, including guidelines for the user. Additionally, we plan to extend the approach to include reverse engineering: We plan to derive ADVERT models from existing code and adapt the existing code to match the code structures necessary for our approach. For validation we plan to experimentally use our approach with evolution scenarios for CoCoME.

Our approach creates additional effort during system development, as existing artefacts (e.g., architectural models) have to be extended and additional artefacts need to be created (e.g., trace link and intermediate models). However, we believe that this effort would pay off during system evolution of the long-living systems, which are target of our approach. We plan to conduct an empirical validation of this hypothesis and to define applicable metrics.

## Acknowledgement

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution.

# 5. REFERENCES

[1] M. Balz. *Embedding Model Specifications in Object-Oriented Program Code: A Bottom-up Approach for Model-based Software Development.* PhD thesis, Universität Duisburg-Essen, 2011.

[2] B. H. C. Cheng and J. M. Atlee. Research directions in requirements engineering. In *Future of Software Eng. (FOSE)*, 2007.

[3] Z. Durdik, B. Klatt, H. Koziolek, K. Krogmann, J. Stammel, and R. Weiss. Sustainability guidelines for long-living software systems. In *Int. Conf. on Softw. Maintenance (ICSM)*, 2012.

[4] Z. Durdik and R. Reussner. Position Paper: Approach for Architectural Design and Modelling with Documented Design Decisions (ADMD3). In *Int. Conf. on the Quality of Softw. Archi (QoSA)*, 2012.

[5] Z. Durdik and R. Reussner. On the Appropriate Rationale for Using Design Patterns and Pattern Documentation. In *Int. Conf. on the Quality of Softw. Arch. (QoSA)*, 2013.

[6] I. Galvao and A. Goknil. Survey of traceability approaches in Model-Driven Engineering. In *Int. Ent. Distributed Object Comp. Conf. (EDOC)*, 2007.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley, 1995.

[8] H. Garbe, C. Janssen, C. Moebus, H. Seebold, and H. de Vries. KARaCAs: Knowledge Acquisition with Repertory Grids and Formal Concept Analysis for Dialog System Construction. In *Managing Knowledge in a World of Networks.* Springer, 2006.

[9] D. Garlan and B. R. Schmerl. Ævol: A tool for defining and planning architecture evolution. In *Int. Conf. on Soft. Eng. (ICSE)*, 2009.

[10] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, M. Meisinger, and C. Pfaller. *The Common Component Modeling Example*, volume 5153 of *LNCS*, chapter CoCoME. Springer, 2008.

[11] M. Jackson. *Problem Frames: Analysing & Structuring Software Development Problems.* Addison-Wesley Professional, 2000.

[12] M. Konersmann and M. Goedicke. A Conceptual Framework and Experimental Workbench for Architectures. In *Softw. Service and Appl Eng.*, volume 7365 of *LNCS*. Springer, 2012.

[13] P. Kruchten. An Ontology of Architectural Design Decisions in Software Intensive Systems. In *2nd Groningen W. Software Variability*, 2004.

[14] M. Müller, M. Balz, and M. Goedicke. Representing Formal Component Models in OSGi. In *S. Eng.*, 2010.

[15] M. Müller, M. Balz, and M. Goedicke. Enriching Java Enterprise Interfaces with Formal Sequential Contracts. In *3rd W. on Behavioural Modelling*, 2011.

[16] J. Stammel, Z. Durdik, K. Krogmann, R. Weiss, and H. Koziolek. Software Evolution for Industrial Automation Systems: Literature Overview. Karlsruhe Reports in Informatics, 2011.

[17] A. Tang, M. Babar, I. Gorton, and J. Han. A survey of the use and documentation of architecture design rationale. In *Conf. on Softw. Arch. (WICSA)*, 2005.

[18] A. Tang, Y. Jin, and J. Han. A rationale-based architecture model for design traceability and reasoning. *J. Syst. Softw.*, 80, 2007.

[19] W. F. Tichy. A catalogue of general-purpose software design patterns. In *Tools-23*, 1997.

[20] W. Wang and J. E. Burge. Using rationale to support pattern-based architectural design. In *ICSE W. on Sharing and Reusing Arch. Knowl.*, SHARK, 2010.