Static Evaluation of Software Architectures

Jens Knodel[1], Mikael Lindvall[2], Dirk Muthig[1], and Matthias Naab[1]

[1] *Fraunhofer Institute for Experimental Software Engineering (IESE), Fraunhofer-Platz 1, D-67663 Kaiserslautern, Germany*
{knodel, muthig, naab}@iese.fraunhofer.de

[2] *Fraunhofer USA Center for Experimental Software Engineering Maryland (FC-MD), 4321 Hartwick Road, College Park, MD USA*
mikli@fc-md.umd.edu

Abstract

The software architecture is one of the most crucial artifacts within the lifecycle of a software system. Decisions made at the architectural level directly enable, facilitate, hamper, or interfere with the achievement of business goals, functional and quality requirements. Architecture evaluations play an important role in the development and evolution of software systems since they determine how adequate the architecture is for its intended usage. This paper summarizes our practical experience with using architecture evaluations and gives an overview on when and how static architecture evaluations contribute to architecture development. We identify ten distinct purposes and needs for static architecture evaluations and illustrate them using a set of industrial and academic case studies. In particular, we show how subsequent steps in architecture development are influenced by the results from architecture evaluations.

Keywords: ADORE, architecture, architecture evaluation, product line, PuLSE-DSSA, reverse engineering.

1. Introduction

One of the most important artifacts in the life cycle of a software system is the architecture since it embraces the decisions and principles for the system to be developed. The software architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution [14]. The goal of an architecture development method is to address such aspects and to provide the fundament for achieving organizational and business goals as well as meeting the functional and quality requirements of the system. To ensure the achievement of these goals, it is mandatory to integrate quality engineering activities into the architecture development method. This is especially true for software product lines [10] since the product line architecture embraces the decisions and principles for each family member. Static architecture evaluation is a sound instrument for assessing and ensuring architectural quality. Static architecture evaluations compare the planned architecture, as described by architectural artifacts, with the actual architecture, as implemented in source code, based on a mapping between the two representations, which is conducted by an expert.

In this paper, we demonstrate the integration of static architecture evaluation into our architecture development method, PuLSE-DSSA, and identify ten distinct purposes and needs for conducting static architecture evaluations. The results of such an evaluation influence and determine subsequent architecture development, which we show in a set of industrial and academic case studies where we applied Fraunhofer PuLSE™ (Product Line Software Engineering) [4] and Fraunhofer ADORE™ (Architecture- and Domain-Oriented Re-Engineering)[1].

The remainder of the paper is structured as follows: Section 2 presents Fraunhofer's PuLSE and ADORE approach. Then section 3 discusses static architecture evaluations and introduces the ten distinct purposes. Section 4 illustrates the purposes and needs in nine case studies. Section 5 then discusses related work, while section 6 concludes this experience report.

2. Approach

The case studies combine two methods: Fraunhofer PuLSE, in particular its architectural component PuLSE-DSSA, and ADORE for reverse engineering activities. This section presents an overview of the two methods and how they relate to the following typical phases of the software life cycle (see Figure 1): architecture development, component engineering, implementation, and finally reverse engineering. Other software development phases (e.g., requirements engineering, testing), feedback cycles and iterations are left out. Next

[1]PuLSE and ADORE are registered trademarks of Fraunhofer Institute for Experimental Software Engineering (IESE), Kaiserslautern, Germany

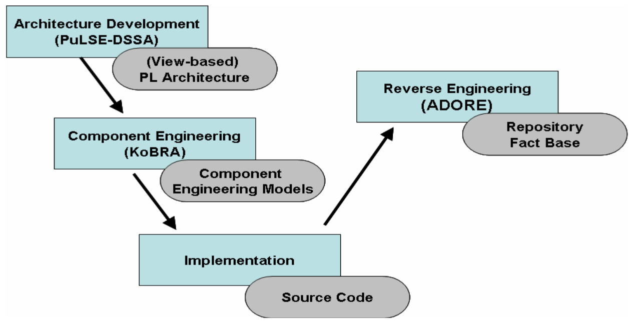
Architecture Development (PuLSE-DSSA)
(View-based) PL Architecture
Component Engineering (KoBRA)
Component Engineering Models
Implementation
Source Code
Reverse Engineering (ADORE)
Repository Fact Base

Figure 1: Lifecycle Phases

to each phase in the figure, the main artifact produced is named.

2.1. PuLSE™-DSSA

PuLSE-DSSA (DSSA stand for Domain-Specific Software Architectures) deals with product line activities at the architectural level (it can also be applied in single system development). Since Greenfield scenarios [10] are rare in industrial contexts, PuLSE-DSSA is designed to smoothly integrate reverse engineering activities to exploit existing knowledge. The main concepts are:

- Scenario-based development in iterations in order to explicitly address stakeholders' needs.
- Incremental development in order to successively prioritize and realize requirements.
- Direct integration of reverse engineering activities into the development process in order to conduct such activities on demand.
- View-based documentation in order to support the communication among stakeholders.

The main process loop of PuLSE-DSSA consists of four major steps (see left part of Figure 2):

Planning: The planning step defines the content and delineates the scope of the current iteration. This includes the selection of a limited set of scenarios that are currently addressed, the identification of relevant stakeholders and roles, the selection and definition of the architectural views to be used, as well as the definition of whether or not an architecture assessment is included at the end of the iteration.

Realization: In the realization step, solutions are selected and design decisions made in order to fulfill the requirements derived from the scenarios. When applying the selected solutions, an implicit assessment regarding the suitability of the solutions for the given requirements and their compatibility with decisions of earlier iterations is made. A catalog of means and patterns is used in this step. Means are principles, techniques, or mechanisms that facilitate the achievement of certain qualities in an architecture whereas patterns are concrete solutions for recurring problems in the design of architectures.

Documentation: This step documents architectures by using an organizational-specific set of views. It relies on standard views as, for example, defined by Kruchten [20] Hofmeister [13], and customizes or complements them by additional aspects as requested by key stakeholders [6].

Assessment: The goal of the assessment step is to analyze and evaluate the resulting architecture with respect to functional and quality requirements and the achievement of business goals. In an intermediate state of the architecture, this step might be skipped and the next iteration immediately started.

PuLSE-DSSA results in (product line) architectures documented in a selection of architectural views.

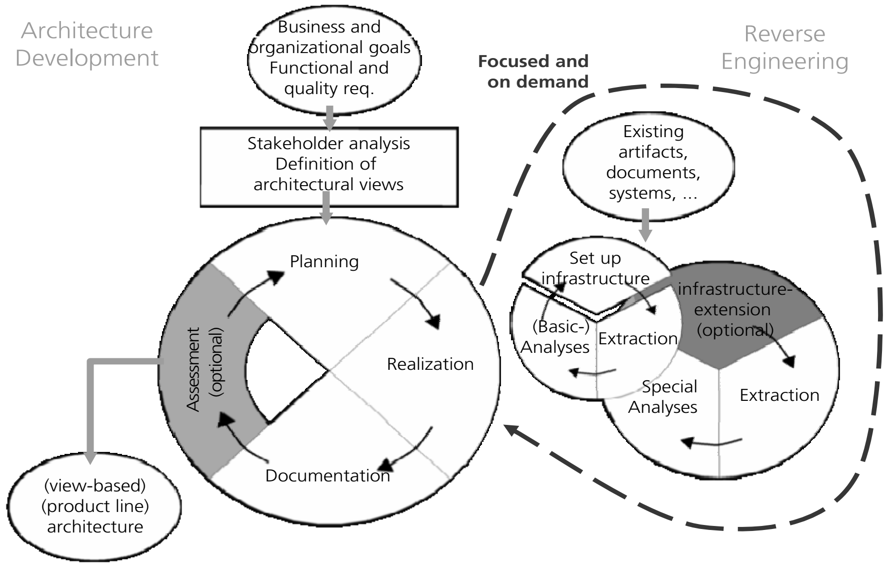
Architecture Development
Business and organizational goals Functional and quality req.
Stakeholder analysis Definition of architectural views
Planning
Realization
Documentation
Assessment (optional)
(view-based) (product line) architecture
Focused and on demand
Reverse Engineering
Existing artifacts, documents, systems, ...
Set up infrastructure
(Basic-) Analyses
Extraction
infrastructure-extension (optional)
Special Analyses
Extraction

Figure 2: Overview PuLSE-DSSA (left side) and ADORE (right side)

2.2. Component Engineering

The architectural views are systematically mapped to models used by Fraunhofer's KoBRA method for engineering component-based product lines [1], [21]. The structural view, for instance, maps to the component containment tree, dynamic views map to interaction models of system or subsystem components. According to Fraunhofer's method, components are modeled by using the Unified Modeling Language (UML) and consist of specification and realization models. Each specification consists of a structural, a behavioral, and a functional model. Each realization consists of a refined structural model, an activity, and an interaction model. Different components can be engineered concurrently since the product line architecture has defined the component communication, specified the required interfaces, and distributed the responsibilities among the components.

2.3. Implementation

The implementation comprises all activities that transform the component engineering models into source code written in a certain programming language. This involves the realization of functionality described in the models, the creation and optimization of the algorithms required to solve computation problems, and restructuring and refactoring of the implemented parts to avoid quality problems like code clones, high complexity of the implementation, or large routines with respect to lines of code.

2.4. ADORE™

The architecture development yields product line components that have to be engineered. In presence of existing systems, this allows the identification of existing components that already completely or partially fulfill the requirements. To decide about reusing such existing components, the component's internal quality and suitability for the product line have to be evaluated. It has to be ensured that the component is able to serve the product line needs.

ADORE™ (Architecture- and Domain-Oriented Reengineering) is a request-driven reengineering approach that evaluates existing components with respect to their adequacy and integrates selected components into the product line:

- First, existing components are identified and reverse engineered [8] to assess their adequacy. This activity is initiated by requests from the product line architects.
- Second, based on the analysis results, the product line architects decide whether the existing component should be reused or a new product line component should be developed from scratch.
- Finally, when reusing the component, necessary renovation and extension activities are conducted in order to adapt the component for its use within the product line.

ADORE is mainly instantiated in step 2 of PuLSE-DSSA (realization), when the architects reason about whether or not to reuse existing components. The architectural needs drive the selection of appropriate reverse engineering analyses. Analyses and, potentially, renovation activities are conducted asynchronously to the PuLSE-DSSA iterations. That is, the current iteration of the architecture development may proceed even if the ADORE activities are delayed. The advantage of such a demand-driven approach is that the investment in reverse engineering is kept as small as possible.

To enable stakeholders to reason about such decisions, certain aspects of the component have to be lifted to a higher level of abstraction. Existing component artifacts (e.g., source code, documentation) are exploited and the information extracted is aggregated into a repository. Since the repository usually has much content, relevant information is often hidden in detailed low-level models. Thus, further analysis activities process the information and aim at creating meaningful views of the existing component.

Typical goals of the reverse engineering process of ADORE address the evaluation of the internal quality of a component, the degree of variability support, the provided flexibility towards anticipated changes, the compliance of a component to the product line architecture, or in case there are several similar implementations of a component, the identification of commonalities among them.

3. Static Architecture Evaluation

Static architecture evaluations compare two models of a software system with each other. Typically, an architectural model (the planned or intended architecture) is compared with a source code model (the actual or implemented architecture), as depicted in Figure 3. Each model consists of a set of (hierarchical) model elements and different types of relations (calls, variable access, etc.) between them. The model elements and the dependencies between them can either be postulated (e.g., an architectural model) or extracted (e.g., a source code model). The comparison requires a mapping between the two models to be compared, which is a human-based task. The comparison assigns one of the following types to each model element and relation between a pair of model elements:

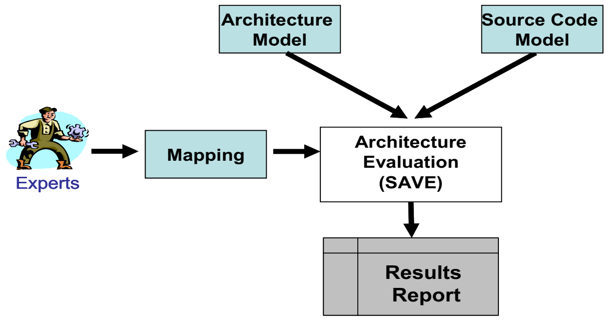

Figure 3: Architecture Evaluations with SAVE

- Convergence – a specific model element or relation exists in both the high level model and in the low level model
- Divergence – a specific model element or relation is only present in the low level model
- Absence – a specific model element or relation is only present in the high level model

The outcome of the evaluation is summarized and documented in a results report (graphical and textual), which can be processed further. The results show whether or not the higher level model converges to the lower level model. The architects can interpret the results by the total numbers of convergences, divergences and absences and use them for the different purposes. In some cases, it is necessary to calibrate the evaluations (i.e., refinement of the high level model, the low level model, or the mapping), which means that the evaluation is performed iteratively.

We performed all evaluations in the case studies described later with the SAVE tool (Software Architecture Visualization and Evaluation), which is based on the Reflexion model idea of [24] and [19]. The SAVE tool is an Eclipse plug-in that is described in [23] and [25].

3.1. Purposes of Static Architecture Evaluation

A static evaluation of software architectures can be performed for different distinct purposes or needs. The purposes differ in their objectives, the reasons why to conduct a static architecture evaluation, and how they influence subsequent steps of architecture development. Below, we introduce ten purposes and relate them to the four steps of PuLSE-DSSA (see Figure 4 for the list of purposes, grouped by the PuLSE-DSSA steps):

ID	PuLSE-DSSA Step	Purpose
1.1	**Planning**	Product line potential
1.2		Product alignment
2.1	**Realization**	Reuse potential
2.2		Component adequacy
2.3		Comprehension
3.1	**Documentation**	Consistency
3.2		Completeness
3.3		Re-documentation
4.1	**Assessment**	Evolution control
4.2		Structure

Figure 4: Evaluation Purposes

1.1 **Product line potential**: This is an analysis of several, independently developed, existing systems with the goal of evaluating how well they would fit under an umbrella of one common reference architecture. The results of such an evaluation help determine which systems could become part of the envisioned product line and therefore should be migrated towards the product line architecture (if necessary). Thus, it guides the planning of the first iteration of PuLSE-DSSA in order to assess the product line potential that is already embodied in existing systems.

1.2 **Product alignment**: This is an analysis of a system that is subject to be merged into an already existing product line. The purpose is to compare the system's architecture to the product line architecture in order to assess conformity and to detect differences. This is conducted using the system's documentation in order to put the PuLSE-DSSA planning step on a sound foundation. This analysis determines the modifications required in order to align the system to the product line architecture and supports estimating the effort to implement those modifications.

2.1 **Reuse potential**: The decision whether to reuse (and integrate into a given architecture) a component, an architectural fragment or part within a product line is often a difficult one. The reuse candidate typically can not be used as is, since it was not designed and realized for the product line, so it is questionable whether it fits the architecture. The degree of dependencies between the reuse candidate and other components of the system and the need for adaptation determine the cost of using the reuse candidate and should drive the reuse decision. Static architecture evaluations are used to visualize the dependencies and the position of the reuse candidate within a decomposition hierarchy. This information helps the product line architects derive sound architectural decisions. In case there are several reuse candidates to the same design problem (see [16] for an example of a comparison approach) the architect can select the best candidate.

2.2 **Component adequacy**: This static evaluation aims at uncovering the internal quality of the subject component. When a component fits the architecture, it is crucial to analyze the component's internal structure and quality. The scope is narrowed down to a single component and its context only. The component is analyzed in depth (usually combined with other reverse engineering techniques like clone detection, variability analysis, code metrics, etc.). The component's internal design, the internal quality, and the component's internal structure are reviewed to assess its adequacy. The component engineering models, including interfaces, are compared to the component's implementation. The results enable the architects to reason about the component's internal adequacy (see [17] for an example).

2.3 **Comprehension**: Program comprehension aims at achieving an understanding of a software system on a high level of abstraction (i.e., an architectural level or a component level). For this purpose, mental models are reflected against the implementation and iteratively improved (e.g., as described in [15] or [24]) until the mental model and the implementation converge. Bottom-up strategies thereby aim at abstracting the models more and more, while top-down strategies refine abstract (domain) views until they conform with the details of the implementation. Static architecture evaluations can be used to achieve better program comprehension by replacing the planned architecture with the mental model and compare it to the model of the implementation.

3.1 **Consistency**: This is an assessment of the degree of consistency of documentation (e.g., architectural views, component engineering models) with the implementation. This analysis determines whether the documentation is still a valid snapshot of the implementation, or if the system and its documentation evolved in different directions. Thus, it aims at increasing the up-to-dateness of the documentation and the degree of consistency.

3.2 **Completeness**: This is an analysis with the goal to detect not yet documented architectural entities (model elements or relations). A key criterion for documentation quality is its completeness. Static evaluations are able to reconcile the elements documented in architecture descriptions with those that are implemented. The identified gaps can now be documented.

3.3 **Re-documentation**: When re-documenting a software system or product line, static evaluations are useful in order to extract the static decomposition (on a low level). Clustering techniques (e.g., see [18]) can then lead to high level architectural descriptions.

4.1 **Evolution control**: Static architecture evaluations are one good means to monitor the evolution of a system or a product line. In addition, they give the architects the possibility to intervene when necessary. After computing a baseline, the focus of an architecture evaluation is set only on the delta (i.e., the modifications made to the system after the baseline was set). The delta viewpoint focuses only on new effects. Another scenario is to define a target architecture, and to evaluate the progress in reaching this target, when modifying the system over time.

4.2 **Structure**: When conducting an assessment, one discussion aspect might be the decomposition and/or the traceability from the architecture to the source code. This discussion can be backed up with the results of the static architecture evaluation.

Purpose/Need	**CS 1**	**CS 2**	**CS 3**	**CS 4**	**CS 5**	**CS 6**	**CS 7**	**CS 8**	**CS 9**
Product line potential								X	
Product alignment					X			X	
Reuse potential									X
Component adequacy							X		
Comprehension	X								
Consistency		X				X			
Completeness						X			
Re-documentation				X					
Evolution control			X	X	X				
Structure	X					X			

Figure 5: Case Study Overview

4. Case Studies

This section presents 9 case studies where we conducted static architecture evaluations in the context of PuLSE-DSSA and ADORE. Figure 5 summarizes the different purposes and needs of the case studies.

4.1. CS1: Apache Tomcat

Subject	Apache Tomcat
Domains	Web server
Type	Academic single system
Language	Java
Size	~ 300 KLOC
Purposes	Comprehension, structure

Figure 6: CS1 - Overview

In [25] an experiment was conducted for the validation of the SAVE visualization component. The hypothesis was that a well-configured visualization of software architectures can support the comprehensibility and the reduction of complexity. In the context of the experiment, we searched for reasonable, realistic tasks concerning an existing system. Apache Tomcat, an open source web server, [2] as being a system of appropriate size (411 classes) was selected as the analysis object for the experiment.

In order to prepare the experimental tasks we (as the experiment designers) had to first understand the Apache Tomcat system ourselves. Due to the trade-off between time constraints and realistic tasks, we had to compose the experiment carefully. We applied our ADORE approach to analyze Apache Tomcat to understand the

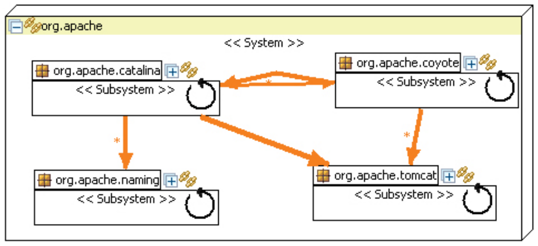

Figure 7: Apache Tomcat High Level View

decomposition structure as well as some of the internal details that would become part of the experimental tasks. The Apache Tomcat system was decomposed into 42 hierarchically nested components. To reduce the visual complexity of the system under investigation, we collapsed the components. Figure 7 presents a high level view of the system and the main subsystems in a UML oriented notation. Starting from this view, we were able to identify which components are related to each other. If components are collapsed, all relations of contained components are lifted to the displayed level.

The results were, for example, that we were able to identify a cyclic dependency between the components org.apache.catalina and org.apache.coyote.

Further analyses of the system required navigating into collapsed components by expanding them in the visualization. This enabled the exploration of the structure of the system in a top-down manner. This strongly supports the comprehensibility of the visualization, as users can decide, what information they want to see. Figure 8 zooms into org.apache.tomcat in order to further explore the internals of that component. Local details of a component are shown while the global context is preserved (i.e., the top-level components stay visible, but their relations point to the low-level components).

An interesting observation, which became an analysis task in the experiment, is concerned with the processing of the TCP/IP and the HTTP communication protocols. While TCP/IP is a stateful protocol with established connections, HTTP is stateless. The extracted facts of the Apache Tomcat exactly reflect this: The TCP/IP protocol,

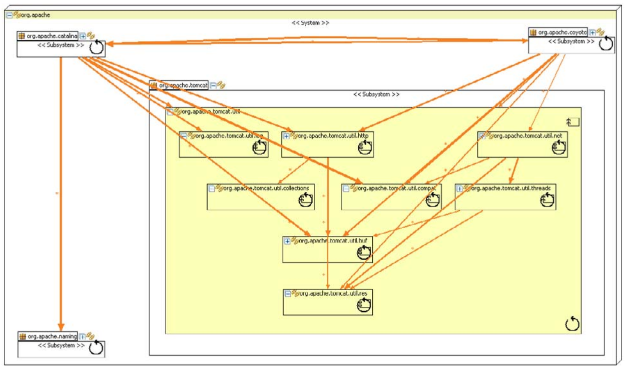
org.apache
<< System >>
org.apache.catalina
org.apache.coyote
org.apache.tomcat
org.apache.tomcat.util
org.apache.tomcat.util.http
org.apache.tomcat.util.net
org.apache.tomcat.util.collections
org.apache.tomcat.util.compat
org.apache.tomcat.util.threads
org.apache.tomcat.util.buf
org.apache.tomcat.util.res
org.apache.naming

Figure 8: Zooming into a Subcomponent

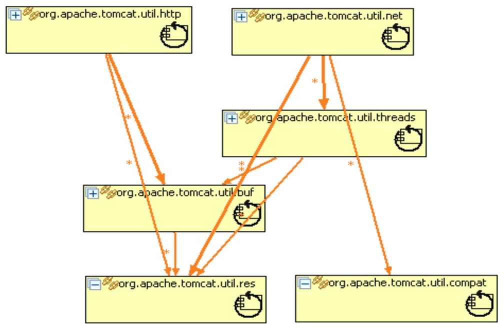
org.apache.tomcat.util.http
org.apache.tomcat.util.net
org.apache.tomcat.util.threads
org.apache.tomcat.util.buf
org.apache.tomcat.util.res
org.apache.tomcat.util.compat

Figure 9: Extraction of the Communication Processing

processed by the org.apache.tomcat.util.net component, uses threads for managing a number of connections initiated by clients. In contrast, the processing of HTTP does not use threads, as requests can be independently processed. Figure 9 presents an extraction of the parts involved in the protocol processing. This shows that Apache Tomcat is a well-structured system, where the naming of components indicates important implementation details.

4.2. CS2: GoPhone

Subject	(Hypothetical) mobile phone
Domains	Demonstrator, mobile phones
Type	Academic product line
Language	Java (J2ME)
Size	~ 10 KLOC per system
Purposes	Consistency

Figure 10: CS2 – Overview

The GoPhone product line is a hypothetical product line of mobile phone implemented in Java. It was developed at IESE as a test bed and demonstrator especially to validate and illustrate product line methods, techniques, or tools [22]. We applied PuLSE-DSSA to

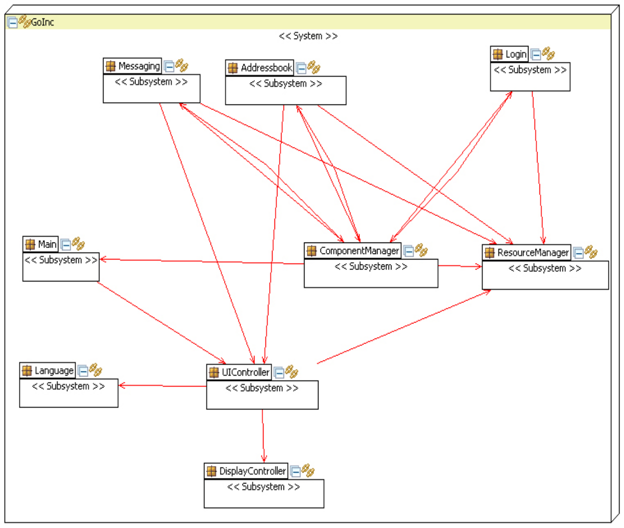

Figure 11: GoPhone Architecture

design architectural modifications of GoPhone, which then were realized by student workers.

Static architecture evaluations helped us assess the degree to which the student workers adhered to architecture guidelines and decisions we made. Thus, the architectural models (see Figure 11 for an overview) developed in PuLSE-DSSA iterations were compared to the source code models implemented to ensure consistency. Since we used the GoPhone product line as a test bed, it was crucial that the concepts we wanted to demonstrate were realized in a clean and smooth manner. We included the static evaluations in the acceptance procedure for the students, and derived to-do lists for refactorings based on the results.

4.3. CS3: SAVE

Subject	SAVE
Domains	Reverse engineering, program analysis
Type	Academic single system
Language	Java
Size	~ 20 KLOC
Purposes	Evolution control

Figure 12: CS3 - Overview

In CS3, we evaluated the implementation of the SAVE tool itself against its architecture description. The tool had to undergo some major restructurings because of some technical constraints that came along with the decision to realize the SAVE tool as an Eclipse plug-in and reuse functionality of other plug-ins (e.g., EMF [11] for persistence, GEF [12] for graphical output).

The purpose of this evaluation was to track the evolution and to evaluate the impact of the constraints. The SAVE tool was hierarchically decomposed into the three plug-ins and several lower level components. Most notable in the evaluation was the absence of relations between the SAVE core and the visualization plug-in. The reason for this was the event propagation mechanism of Eclipse that has no static dependencies (see [23] for a detailed description). We updated our architectural description accordingly and paid special attention to the event propagation mechanism in the further evolution of SAVE.

4.4. CS4: TSAFE

Subject	Air traffic control system
Domains	Demonstrator, air traffic control
Type	Academic single system
Language	Java
Size	~ 20 KLOC
Purposes	Re-documentation, evolution control

Figure 13: CS4 - Overview

TSAFE is a test bed to aid air-traffic controllers in detecting and resolving short-term conflicts between aircrafts [26]. FC-MD used TSAFE in order to conduct an experiment about the preservation of software architecture flexibility when unfamiliar developers change the system due to new requirements [1]. Before the experiment, static architecture evaluation helped in the re-documentation of the system when the architectural flexibility concepts were built-in.

In addition, the SAVE tool helped analyze and visualize parts of the experiment's results. The goal of the experiment was to assess the ease of the introduction of new requirements to the TSAFE system, depending on the quality of the software architecture of the system. The subjects were split into seven teams and the teams were divided into two groups – one group worked on a system with an architecture with built-in flexibility and the other one worked on the original system without built-in flexibility. Both groups had the task to extend the system to fulfill new requirements, the same for each group. The SAVE tool monitored the results of the students focusing only on the deltas introduced after starting the analysis.

4.5. CS5: Migration to a Reference Architecture

Subject	Car window opener
Domains	Embedded system, car electronics
Type	Industrial single systems
Language	C
Size	~ 10 – 20 KLOC per system
Purposes	Product alignment, evolution control

Figure 14: CS5 - Overview

A customer and IESE applied the PuLSE-DSSA method to design a product line architecture for an existing family of car window openers, where each system differs slightly from the others because of the different car manufacturers' requirements. One system was refactored according to the architecture designed and we removed some architectural flaws, and then we extended the architecture to product line needs. The refactorings were monitored by static architecture evaluation to show the progress of the refactorings, the distance to the final state, and to prove that the changed implementation really is compliant to the reference architecture. The final result was a layered architecture allowing only strict top-down dependencies, the only exceptions were some callbacks violating the layered structure and some include dependencies due to configuration files and third party software (which were not in the scope of the case study).

Ongoing work will restructure further systems of the window opener family compliant to the product line architecture. The PuLSE-DSSA cycles conduct static architecture evaluation for the product line alignment purpose to decide whether to include the system in the product line and to estimate the effort required to align it.

4.6. CS6: Product Line versus Implementations

Subject	Climate measurement devices
Domains	Embedded system, measurement
Type	Industrial product line
Language	C
Size	~ 200 – 600 KLOC per system
Purposes	Consistency, completeness, structure

Figure 15: CS6 - Overview

In CS6, three members of a product line of climate measurement devices were derived from a product line infrastructure that provides a framework with generic components. The task was to assess the consistency between the product line architecture and the three derived products as well as to check the completeness of the architecture documentation. The outcome was an action list where to adopt the architecture, and how to better support the derivation of future products with the help of the infrastructure. Figure 17 presents, as an example of the architecture evaluations, a table with call dependencies between one product and the framework. The dependencies are either internal ones (calls within the product, or in the framework) and external ones (from the product to the framework, or vice versa). By zooming into the decomposition hierarchy, the architects were able to learn about the affected components, files, functions and variables.

One result was that an unexpected high number of dependencies of call relations from the framework-related source code to the product code were detected. Such communication between the framework and the products is a major risk to the structure of the framework since framework functionality relies on product implementation. Furthermore, the results show that the underlying layered architecture was significantly violated not only by call relations but also by includes, and variable accesses. So the structural decomposition was degenerated and the documentation was neither consistent nor complete. The action items derived were concerned with the adaptation of either the reference architecture or the product implementation. It was recommended to refactor obsolete dependencies (e.g., includes, but no included element is used), as well as to encapsulate global variables. Another action item is a reconsideration of the current interfaces between components. These quality issues were addressed in the next PuLSE-DSSA cycle.

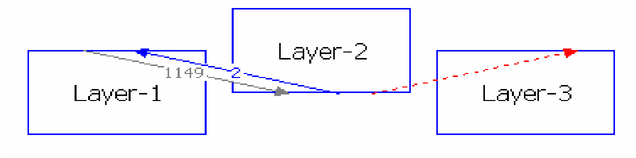

Figure 16: Layers

CALLS →	CALLEE		Total
CALLER	Product_1	Framework	
Product_1	9226	58	9284
Framework	1021	858	1879
Total	10247	916	11163

Figure 17: Product versus Framework

4.7. CS7: Component Adequacy

Subject	Graphics Component
Domains	Embedded system, car multimedia
Type	Industrial product line
Language	C++
Size	~ 180 KLOC
Purposes	Component adequacy

Figure 18: CS7 - Overview

CS7 deals with the implementation of a first product line component in the context of a migration project where an organization incrementally transitioned from single system development to product line engineering. The component (at the time of the evaluation still under development) was responsible for the graphical output of a car multimedia system on a panel. Since it would become the first product line component, the quality of the implementation was of special interest. The adequacy of the component was statically evaluated with the help of the SAVE tool (next to other analyses like clone detection, variability analysis).

The component engineering models decomposed the subject into the three internal layers. Figure 16 depicts the results of the evaluation. The evaluation shows a high degree of adequacy so far since there are almost no violations of the documented component engineering model (Layer-1 uses Layer-2, cardinality 1149) except two divergences from Layer-2 to Layer-1 (cardinality 2) and one absence from Layer-2 to Layer-3. The reason for the latter is that the component is currently still under development, and this layer has not yet been realized. The implementation so far adhere the intended design decisions, although detailed analysis of the layers gave pointers for improvement. The challenge for the development organization is now to ensure this adherence over time. We recommended continuing the architecture evaluations and monitoring the component's evolution when creating new variants based on this component.

4.8. CS8: Product Line Potential

Subject	Engine control system
Domains	Embedded system, car body electronics
Type	Industrial single systems
Language	C
Size	~ 100 – 500 KLOC per system
Purposes	Product line potential, product alignment

Figure 19: CS8 - Overview

CS8's subject was a project where a development organization already had a couple of similar existing systems. The task was to estimate the product line potential of these related systems. The common reference architecture was available, but it was not valid because the products were developed following the clone-and-own principle. To be able to evaluate the product line potential, we had to understand the system architectures of the existing systems.

We applied the SAVE tool to evaluate the reference architecture against the different product implementations. The goals were to explore the product line potential and to what extent there are commonalities among the existing products, and to gain knowledge whether a product line could be built on top of the given reference architecture. The results of the static evaluations showed the degree to which the existing systems violated the reference architecture (partially to a large extent). Nevertheless, the results revealed some common components among the products bearing high product line potential. This influences the planning of the first PuLSE-DSSA iteration cycle by selecting the scenarios that deal with the common parts first, in order to achieve benefits of product line engineering fast.

4.9. CS9: Commonalities among Products

Subject	Digital camera
Domains	Embedded system
Type	Industrial single systems
Language	C
Size	~ 400 – 600 KLOC per system
Purposes	Reuse potential

Figure 20: CS9 - Overview

Three digital camera systems were the subject of the ongoing case study CS9. The task was to exploit the commonalities among the systems in order to establish a product line infrastructure. All three products were analyzed statically for common parts (i.e., components that were the same from an architectural viewpoint for all three systems). The identified commonalities are subjects to be migrated into a common infrastructure.

One result was the identification of a monolithic block having a lot of dependencies to other subsystems, which caused maintenance problems. We addressed this issue in the PuLSE-DSSA by decomposing the monolithic block into several smaller components with defined interfaces.

5. Related Work

The basic concepts of the SAVE tool are similar to the Reflexion model technique presented by Murphy [24] in comparing an extracted source code model and a high level model created by the user. The computed model is called Reflexion model and shows where the planned high level model agrees with and where it differs from the extracted dependencies of the source code. Koschke [19] extended the Reflexion model to support hierarchies allowing model elements be part of other elements. The SAVE tool supports hierarchies as well.

Postma [26] introduced another method of software architecture verification. This method is based on architectural rules. A rule expresses conditions on multiple relations and therefore this kind of verification is more general than verifying only one relation. The rules are defined using a Relation Partition Algebra (RPA).

Architectural tracking is the process of comparing the specified software architecture of the system and the actual implementation of the system in a regular manner. FC-MD (see [27]) had previously developed an approach for architectural tracking, which is now adapted to the SAVE tool and integrated into the PuLSE-DSSA method.

The software architecture analysis method (SAAM [9]) evaluates the modifiability of software architectures with respect to a set of representative change scenarios. The architecture tradeoff analysis method (ATAM [9]) is also a scenario-based method, which extends SAAM to address further quality attributes. Its goal is to analyze whether the software architecture satisfies given quality requirements and how the satisfaction of these quality requirements trade off against each other.

In [7], Bosch presents four architecture assessment techniques (i.e., scenario-, simulation-, mathematical model- and experience-based assessments). These techniques aim at the evaluation whether a system fulfills its quality requirements or not.

6. Conclusion

Static evaluations of software architectures are a sound instrument to control, learn and assess architectural aspects and implementations of architectures. This experience report presented ten distinct purposes and needs for conducting static evaluations. In 9 case studies (5 industrial and 4 academic), we showed how static architecture evaluations contributed to the further development and evolution of architectures. All case studies were conducted with the help of the SAVE tool, which connects the architectural evaluation activities to the source code. The SAVE tool allows the navigation from components to the code. This strongly supports the comprehension of relationships between source code and architectural models. The case studies described how the results of static evaluations can steer ongoing architectural development by underpinning architectural decisions. It was also demonstrated how architectural evaluations contribute to the successful achievement of functional requirements and quality goals of the overall system or product line.

Up to now, we have applied static evaluations only for single purposes; there was not yet a long-term case study that covered all purposes across all PuLSE-DSSA steps, several iteration cycles and including a long-term evolution of the product line. We plan to address this issue in a future case study.

Ongoing work includes the development of a mechanism that will allow us to perform dynamic evaluations based on runtime scenario traces that can be compared against behavioral models or dynamic architectural views. Another potential extension is to include information of configuration management systems to derive statements about historic trends.

7. References

[1] B.Anders, J. Fellman, M. Lindvall, I. Rus: Experimenting with Software Architecture Flexibility Using an Implementation of the Tactical Separation Assisted Flight Environment, Proceedings of IEEE/NASA SEL Workshop, 2005.

[2] Apache Tomcat *http://jakarta.apache.org/tomcat/index.html*

[3] C. Atkinson et al.: Component-based Product Line Engineering with UML, Addison-Wesley, 2001

[4] J. Bayer et al.: PuLSE: A Methodology to Develop Software Product Lines, 5th Symposium on Software Reusability (SSR'99), 1999

[5] J. Bayer et al: Definition of Reference Architectures based on Existing Systems, (IESE-Report 034.04/E), 2004

[6] J. Bayer: View-Based Software Documentation, PhD, Fraunhofer IRB Verlag, 2004.

[7] J. Bosch: Design & Use of Software Architectures, Addison-Wesley, 2000

[8] E. Chikofsky, and J. H. Cross: Reverse Engineering and Design Recovery: a Taxonomy, IEEE Software, 7(1):13-17, January 1990

[9] P. Clements, R. Kazman., and M. Klein: Evaluating Software Architectures: Methods and Case Studies, Addison-Wesley, 2002

[10] P. Clements and L. M. Northrop: Software Product Lines: Practices and Patterns, Addison-Wesley, 2001

[11] Eclipse Modeling Framework, *http://www.eclipse.org/emf*

[12] Graphical Editing Framework, *http://www.eclipse.org/gef/*

[13] C. Hofmeister, R. Nord, R., and D. Soni: Applied Software Architecture. Addison-Wesley, 1999

[14] IEEE Standard 1471 - Recommended Practice for Architectural Descriptions of Software-Intensive Systems, IEEE Computer Society, 2000

[15] J. Knodel: Reconstruction of Architectural Views by Design Hypothesis, Softwaretechnik-Trends, 2003

[16] J. Knodel, T. Forster, J. F. Girard: Comparing design alternatives from field-tested systems to support product line architecture design, CSMR, March 2005

[17] J. Knodel, D. Muthig: Analyzing Product Line Adequacy of Existing Components, Workshop on Reengineering towards Product Lines (R2PL), November 2005

[18] R. Koschke: Atomic Architectural Component Recovery for Program Understanding and Evolution, PhD, University of Stuttgart, 2000.

[19] R. Koschke, D. Simon: Hierarchical Reflexion Models, Working Conference on Reverse Engineering, 2003, pages 36-45.

[20] P. Kruchten: The 4+1 View Model of Architecture. IEEE Software, November 1995 12(6):42–50.

[21] D. Muthig and C. Atkinson: Model-driven Product Line Architectures. Software Product Line Conference (SPLC2), San Diego, CA, 2002

[22] D. Muthig, et al.: GoPhone - A Software Product Line in the Mobile Phone Domain, Kaiserslautern, 2004 (IESE-Report 025.04/E)

[23] P. Miodonski, T. Forster, J. Knodel, M. Lindvall, D. Muthig: Evaluation of Software Architectures with Eclipse, Kaiserslautern, 2004, (IESE-Report 107.04/E)

[24] G. C. Murphy, D. Notkin, K. Sullivan: Software reflexion models: Bridging the gap between design and implementation. IEEE Transactions on Software Engineering, 27(4), April 2001.

[25] M. Naab, T. Forster, J. Knodel, D. Muthig: Evaluation of Graphical Elements and their Adequacy for the Visualization of Software Architectures, Kaiserslautern, 2005, (IESE-Report 078.05/E)

[26] A. Postma: A method for module architecture verification and its application on a large component-based system, Information & Software Technology 45(4), 2003, 171-194

[27] R. Tvedt, P. Costa, M. Lindvall: Evaluating Software Architectures, Advances in Computers, Elsevier Science, 2004

[28] TSAFE: Tactical Separation Assisted Flight Environment, *http://sdg.lcs.mit.edu/TSAFE/*