Fraunhofer Institut
Experimentelles
Software Engineering

Static Evaluation of Software Architectures

Authors:
Jens Knodel
Mikael Lindvall
Dirk Muthig
Matthias Naab

Submitted for publication at
WICSA5, Working Conference
on Software Architecture

IESE-Report No. 036.05/E
Version 1.0
May 30, 2005

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competitive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach (Executive Director)
Prof. Dr. Peter Liggesmeyer (Director)
Fraunhofer-Platz 1
67663 Kaiserslautern

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 as well as meeting functional and quality requirements. The latter includes reusability, and thus software architectures are also essential for the success of product line engineering.

This paper summarizes our practical experience by giving an overview on when and how static architecture evaluation practically contributes to architecture development. Therefore, it defines ten distinct purposes of architectural evaluations and illustrates them in a set of industrial and academic case studies. Most of the case studies are settled in a product line engineering context. In particular, we highlight how the different purposes determine and influence subsequent steps in architecture development.

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

Table of Contents

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 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 the goals, it is mandatory to have quality engineering activities as an integrated part in an 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. A sound instrument to assess and ensure architectural quality is to conduct static architecture evaluations. 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).

In this paper, we demonstrate the integration of static architecture evaluation into our architecture development method, PuLSE-DSSA and identify ten distinct purposes for conducting static architecture evaluations. The results of such an evaluation influence and determine subsequent architecture development. We demonstrate the impact 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 4 discusses static architecture evaluations and introduces ten distinct purposes for static architecture evaluations integrated into PuLSE. Section 4 illustrates the role of the distinct purposes in nine industrial and academic case studies. Chapter 5 then discusses related work, while chapter 6 concludes this experience report.

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

2 Approach

The case studies combined 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 other phases in the software life cycle (Figure 1 depicts the phases) as typically proposed by software development processes, first architecture development, then component engineering, and the implementation, and finally reverse engineering; other software development phases (e.g., requirements engineering, testing, etc.), potential feedback cycles and iterations are left out. Next to each phase, there is the main artifact produced in it.

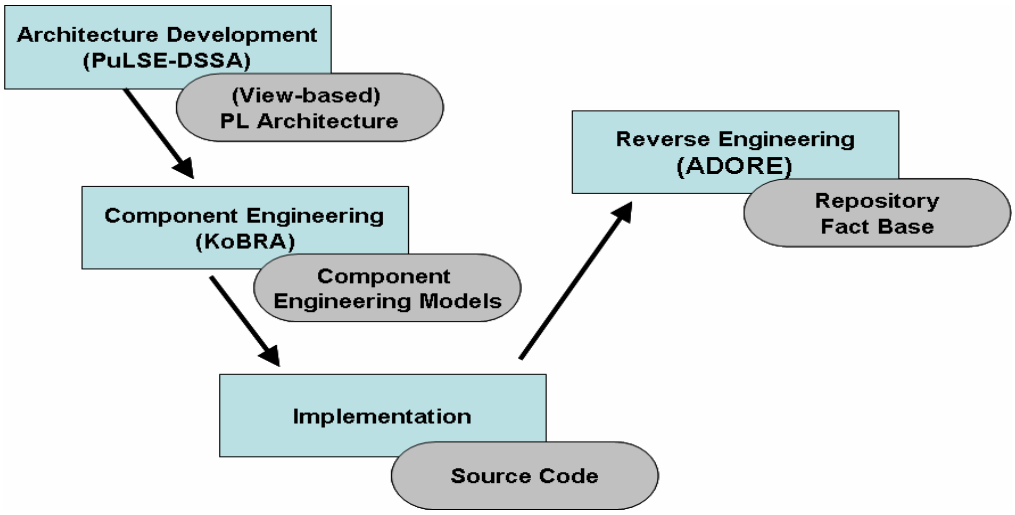

Figure 1: Lifecycle Phases

2.1 PuLSE™-DSSA

PuLSE™-DSSA deals with product line activities at the architectural level (it can also be used in single system development). Since greenfield scenarios 10 are found only rarely in industrial contexts, PuLSE-DSSA is designed to smoothly integrate reverse engineering activities into the process of developing a product line architecture. The main underlying concepts of the PuLSE-DSSA are:

- Scenario-based development in iterations that explicitly addresses the stakeholders' needs.

- Incremental development, which successively prioritizes requirements and realizes them.
- Direct integration of reverse engineering activities into the development process on demand.
- View-based documentation to support the communication of different roles.

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

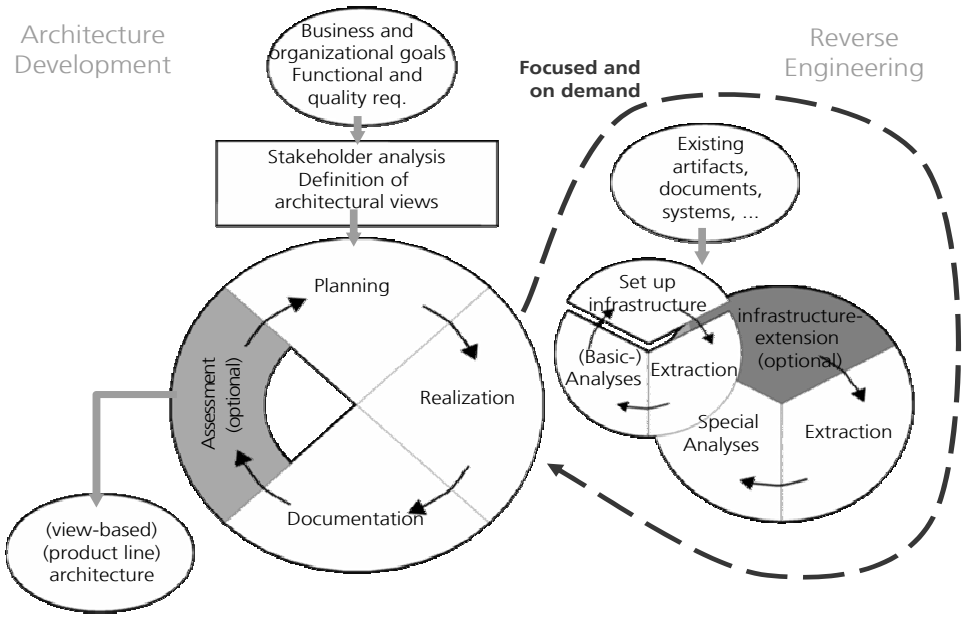

Figure 2: Overview PuLSE-DSSA (left side) and ADORE (right side)

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

Realization: In the realization phase, solutions are selected and design decisions taken in order to fulfill the requirements given by the scenarios. When selecting and applying the selected solutions, an implicit assessment regarding the suitability of the solutions for the given requirements and their compatibility with design decisions of earlier iterations is made. A catalog of means and patterns is used in this phase. 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 thereby relies on standard views as, for example, defined by Kruchten 20 or Hofmeister 13, and customizes or complements them by additional aspects requested by one of the 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 is started.

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

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 a specification and a realization. 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. The different components can then be engineered concurrently since the 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 definition yields in product line components that have to be engineered. In the implementation phase, the components and interfaces are realized. Reverse engineering activities then enable the analysis of the implemented architecture. One goal of reverse engineering as defined in 8 is to create representations at higher levels of abstraction. Therefore, facts are extracted from existing artifacts (e.g., source code, documentation, configuration files) and the information is aggregated in a fact base or repository. Since the fact base contains a large amount of information, the most information is often hidden in overcrowded low-level models. Therefore, further analysis activities process the information and aiming at creating meaningful views of the existing systems.

ADORE™ (Architecture- and Domain-Oriented Reengineering) is a request-driven reverse engineering approach that takes the architecture and the domain context of the analyzed artifacts into account. ADORE is mainly instantiated in step 2 of PuLSE-DSSA (realization), when the architects typically reason about whether or not to reuse existing components. The architecture drives the selection of reverse engineering activities and the results of those activities answer the reuse question. Reverse engineering activities are conducted asynchronously to the PuLSE-DSSA iteration. That is, the current iteration of the architecture development may proceed if the answer to the reverse engineering request is delayed. The advantage of such a request-driven approach is that investment into reverse-engineering is kept as small as possible.

Typical goals of reverse engineering in the context of product line engineering are a) recovery of lost information in order to benefit from field-tested solutions and experiences, b) localization of single features in the source code in order to reuse this functionality in the product line, c) enabling reuse in order to integrate components (or whole subsystems) into the product line, and d) evaluating the architecture for quality assurance purposes.

3 Static Architecture Evaluation

Static architecture evaluations compare two models of a software system with each other. Typically an architectural model (the intended architecture) is compared with a source code model (the 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., in a high level model such as an architectural view) or extracted (e.g., in a low level model such as the source code). 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:

- Convergence – a model element of a relation in the high level model was also present in the low level model
- Divergence – a model element or a relation was present in the low level model, but missing in the high level model
- Absence – a model element or a relation was present in the high level model, but the counterpart is missing in the low level model

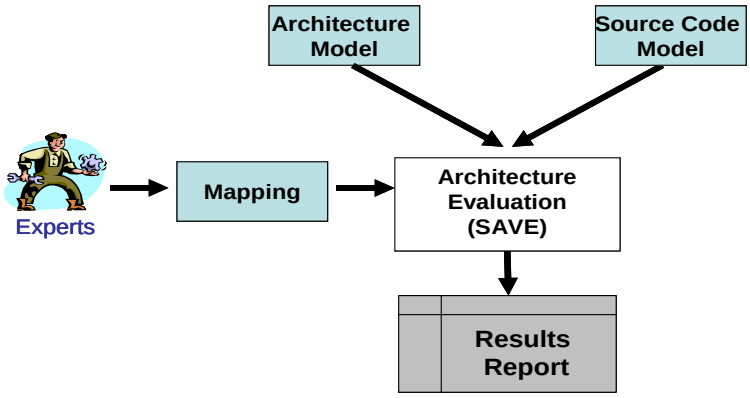

Figure 3: Architecture Evaluations with SAVE

Next, the outcome of an 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 total number of convergences indicates the degree of convergence, while the number of divergences and absences indicate the opposite. The architects can interpret the results (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 more detail in 23 and 25.

3.1 Purposes of Static Architecture Evaluation

A static evaluation of software architectures can be performed for different, distinct purposes. The purposes differ in their objectives, the reasons why to conduct a static architecture evaluation and how they influence subsequent steps of architecture development. In the following we will introduce ten distinct 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**: An analysis into whether several, independently developed, existing systems are realized under an umbrella of one common reference architecture. The results of such an evaluation help to determine which products become part of the envisioned product line and therefore have to 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**: When one system is subject to be merged into an already existing product line, the purpose is to evaluate the product line architecture on the one hand against the system's architecture to assess the conformity and to

detect differences and on the other hand against the system's documentation in order to put the PuLSE-DSSA planning step on a sound foundation. This supports the effort estimation in order to align the system to the product line architecture and analyzes the degree of required 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 not easy to make. 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 to the architecture. The dependencies of the reuse candidate have to be revealed and the need for its (potential) adaptation should drive the reuse decision. Static evaluations help to visualize the dependencies and the position of the reuse candidate within a decomposition hierarchy. This information helps the product line architects to derive sound architectural decisions. In case, there are several reuse candidates to the same design problems, see 16 for an example of a comparison approach.

2.2 **Component adequacy**: This static evaluation aims at uncovering the internal quality of the subject component. The scope is narrowed down to a single component and its context only. It 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 derive statements about the component 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 agree. 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.

3.1 **Consistency**: An assessment of the degree of consistency of documentation (e.g., architectural views, component engineering models) with the implementation. It is checked where the documentation is still a valid snapshot of the implementation, or if both evolved differently. Thus, it aims at increasing the up-to-dateness of the documentation.

3.2 **Completeness**: An analysis in order 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 docu-

mented 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 very useful to extract the static decomposition (on a low level), when combined with clustering techniques (e.g., see 18) then the combination of both can lead to high level architectural descriptions.

3.4 **Evolution control**: Static architecture evaluations are one good means to monitor the evolution of a system or a product line and 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). This filtering emphasizes only new effects (convergences, violations, degeneration). Another scenario is to define a target architecture, and to evaluate the progress in reaching this target, when modifying the system over time.

4.1 **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 static architecture evaluations concerning the structural decomposition of a system.

4 Case Studies

This section presents nine case studies where we applied static architecture evaluations with the SAVE tool in the context of PuLSE-DSSA and ADORE for different purposes (see Figure 5 for an overview).

Purpose	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				X
Component adequacy							X		
Comprehension	X							X	
Consistency		X				X			
Completeness						X			
Re-documentation				X					
Evolution control			X	X	X				
Structure	X					X			

Figure 5: Case Study Overview

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 for 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 2 as being a system of appropriate size (411 classes) was selected as the analysis object for the experiment. Apache Tomcat is an open source web server of the Apache software foundation.

In order to be able 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 decomposition structure and to find out about internal details that become part of the experimental tasks. The Apache Tomcat system could be decomposed into hierarchically nested 42 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. For instance, we were able to identify a cyclic dependency between the components org.apache.catalina and org.apache.coyote.

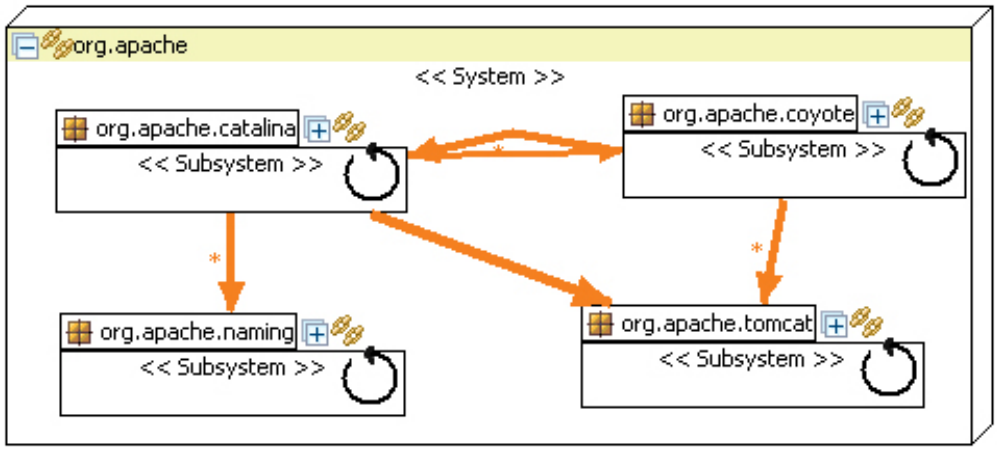

Figure 7: Apache Tomcat High Level View

Further analyses of the system required the navigation 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 zoomed 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).

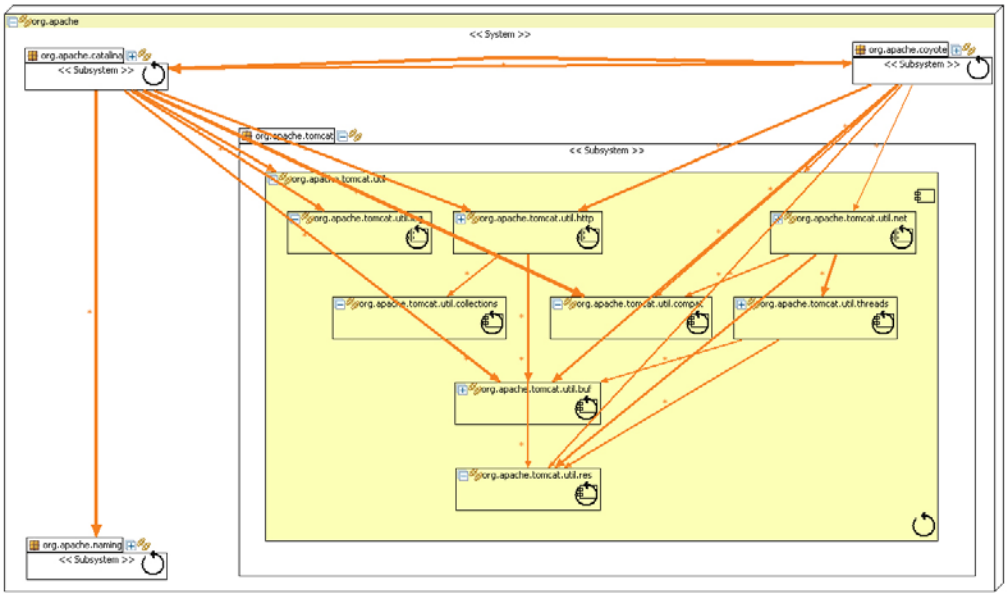

Figure 8: Zooming into a Subcomponent

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, 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.

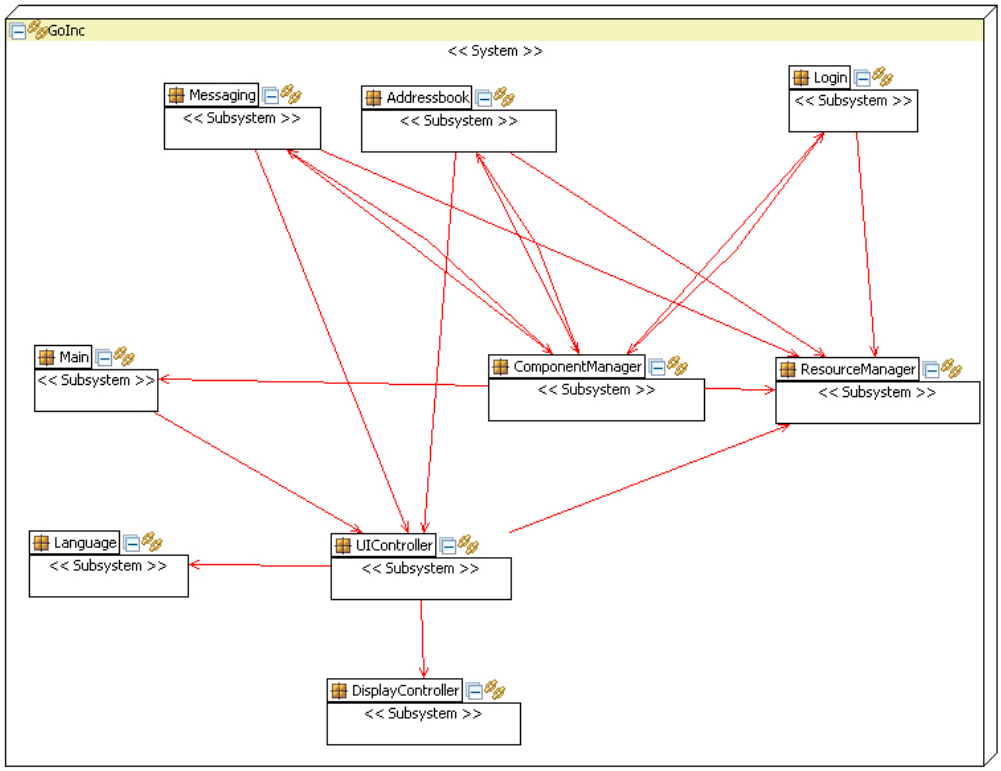

Figure 9: Go Phone Architecture

The SAVE tool connects the architectural evaluation activities to the source code. It is possible to navigate to the source code files belonging to a component directly from the view, furthermore it is also possible to jump to the source and destination of a relation. This strongly supports the comprehension of relationships between source code and architectural models.

4.2 CS2: Go Phone

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.

To ensure the proper mode of operation of the demonstrator, we apply PuLSE-DSSA to design architectural modifications, which are then typically realized by student workers.

Static architecture evaluations help us to assess the degree to which the student workers adhere to architecture guidelines and decisions we made for implementation details. Thus, the architectural models (see Figure 9 for an overview) developed in PuLSE-DSSA iterations are compared to the source code models implemented for the consistency purposes. Since we use the GoPhone product line as a test bed, it is crucial that the concepts we want to demonstrate are realized in a clean and smooth manner. We included the static evaluations in the acceptance procedure for the students, and depending on the results, we either redesign the architecture or the student workers obtain a to-do list to refactor their solutions.

4.3 CS3: SAVE

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

Figure 11: CS3 - Overview

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

The purpose of this evaluation was to track the evolution and in order to be able to reason about the design decisions. The SAVE tool was hierarchically decomposed into the three plug-ins and several lower level components. Most notable in the evaluation were the absence between the SAVE core and the visualization plug-in. The reason for this was the event propagation mechanism of Eclipse, which has no static dependencies. A detailed description of this case study can be found in 23. We updated the architectural description of the SAVE tool accordingly and paid special attention to the event propagation mechanism in the assessment step of PuLSE-DSSA.

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 12: 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 to analyze and to visualize parts of the results of the experiment. 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 the clean, documented architecture and the other one worked on the original system having a messy architecture. 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, reuse potential, evolution control

Figure 13: 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. The case study started with an assessment of the reuse potential of one existing system, and it was decided to use this system as basis for the architecture development. Then we extended the architecture in PuLSE-DSSA cycles

with respect to product line needs and removed some architectural flaws already known for the system.

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 was 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 the system.

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 14: CS6 - Overview

In CS6, three members of a product line of climate measurement devices were derived from a product line infrastructure providing a framework with generic components. The goal of this case study was to assess the consistency between the product line architecture and three derived products, and to check the completeness of the architecture documentation. The intended 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 15 presents as an example of the architecture evaluations a textual overview on the call dependencies between one product and the framework. The dependencies are given internally (call within the product, or in the framework) and external from the product to the framework, or vice versa; the dependencies on the highest level of abstraction, it is possible for the architects to navigate and zoom into the details over the decomposition hierarchy, from the products over architectural layers, to component hierarchies, to files and finally to function, procedures and variables. The communication between framework and products shows an unexpected high number of dependencies of calls from the framework-related source code to product code. This is a major risk to the

structure of the framework, since framework functionality relies on product implementation. Furthermore, the results showed that underlying layered architecture was significantly violated not only by call but as well by includes, and variable accesses. So the documentation was neither consistent nor complete.

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

Figure 15: Product versus Framework

The actions items derived comprise a detailed analysis of the divergences and adaptation of either the reference architecture or the product implementation. Obsolete dependencies (e.g., includes, but no included element is used) are refactored, as well as the reduction of global variables is a goal for restructuring activities (they should be encapsulated). Another action item is a reconsideration of the current interfaces between components. These quality issues were input to another PuLSE-DSSA cycle.

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 16: 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 TFT-panel. Since it should 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).

The component engineering models decomposed the subject into the three internal layers. Figure 17 depicts the results of the evaluation. The evaluation shows a high degree of adequacy so far since there are almost no violations to the documented component engineering model (Layer-1 uses Layer-2, grey ar-

row, cardinality 1149); there are only two exceptions: the divergences from Layer-2 to Layer-1 (blue dashed arrow, cardinality 2) and the 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 evaluation shows that the implementation so far did follow 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 over time. The component's evolution should be monitored when new variants are created based on this first product line component. To keep the quality and to avoid degeneration, we recommended quality assurance activities including the continuous architectural evaluations.

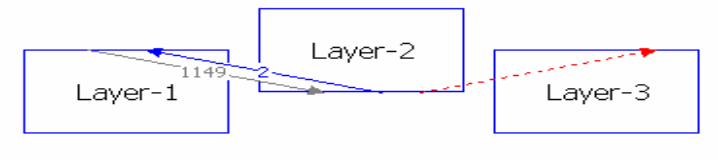

Figure 17: Layers

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, comprehension

Figure 18: CS8 - Overview

CS8's subject was a project where a development organization has already a couple of existing products. The question was to estimate the product line potential of the related systems. A common reference architecture was documented, more or less valid because the products were developed following the clone-and-own principle. To be able to state the product line potential, we had to understand the system architectures given. So we conducted a couple of analysis activities, one of them was 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 that the systems were degenerated, partially to a large extent. Nevertheless, the results identified some commonalities among the products bearing high product line potential. This influences the planning of the first PuLSE-DSSA

iteration cycle by selecting the set of scenarios that deals 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 19: CS9 - Overview

Three digital camera systems were the subject of the ongoing case study CS9. In order to exploit the commonalities among the systems 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 commonalities identified are subjects to be migrated into a common infrastructure. Static architecture evaluations indicate the potential that exists among those products and enables the architects to estimate potentials saving due to reuse for future derived products. Further architecture development will include variability implementation techniques to manage the differences among the systems.

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. Thereby it is allowed that a high level model element to be part of another 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, to learn, and to assess architectural aspects and its implementation. This work presents ten different, distinct purposes for conducting static evaluations and how the results serve different goals within our architecture development method PuLSE-DSSA. We demonstrated in this experience report how such evaluation influences the further architectural development and presented nine case studies (5 industrial and 4 academic), where we exemplified how static architecture evaluations contributed to the architecture-driven development. All case studies were conducted with the help of the SAVE tool. The results of conducted static evaluations steer ongoing architectural development by underpinning architectural decisions and thus, they contribute to the successful achievement of functional requirements and quality goals of the overall system or product line.

Up to now, we applied static evaluations only for limited 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 want to address this issue in future case studies.

Ongoing work will include a mechanism to be able to perform dynamic evaluation based on runtime scenario traces, which then can be compared against behavioral model or dynamic architectural views. Another potential extension is to include version histories (when the system is under control of a configuration management system) to be able to make statement about historic trends and development directions.

7 References

1. Anders B., Fellman J., Lindvall M., and Rus I.: 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, Submitted to 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

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, 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 source and high-level models, ACM Software Engineering Notes, 1995

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, 2003

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/*

Document Information

Title:	Static Evaluation of Software Architectures
Date:	May 30, 2005
Report:	IESE-036.05/E
Status:	Final
Distribution:	Public