Static Architecture Evaluation of Open Source Reuse Candidates

Jens Knodel, Dirk Muthig, and Matthias Naab
Fraunhofer Institute for Experimental Software Engineering (IESE)
Fraunhofer-Platz 1, D-67663 Kaiserslautern, Germany
{knodel, muthig, naab}@iese.fraunhofer.de

Abstract. Open source software systems provide a variety of field-tested components offering software development organizations the potential to reuse and adapt such components for their own purposes. The main challenge before achieving the reuse benefits is to acquire a thorough understanding of open source software systems (i.e., the reuse candidates) in order to reason about alternative solutions, to learn about the points where to adapt the system and eventually to decide whether or not to invest into reuse. Manually analyzing even small systems is a time-consuming, complex and costly task. In this paper we present a case study where we analyzed the Apache Tomcat web server supported by a software architecture visualization and evaluation tool and demonstrate how the tool facilitated our comprehension tasks to learn about the architectural means and concepts.

Keywords. Apache Tomcat, architecture evaluation, Eclipse, open source system, SAVE, software architecture, static analysis, visualization.

1 Introduction

Open source software allows software development organizations to benefit from existing, field-tested solutions where the code base is available at no cost. Assembling open source software into the own developments can avoid the "reinventing the wheel" syndrome. There are two main strategies for the integration of open source software into the development:

- *Wrapping*: the open source software is used as a black-box, and the developers only need to know about the interfaces, the open source software itself is not changed at all
- *Adaptation*: the open source software is used as a white box and the developers modify the open source software where necessary to adapt it to their needs. Detailed knowledge about the architecture and the implementation underlying the open source system is required.

Pizka [Pi04] reports a case where "the adaptation of existing software has demonstrated its superiority compared to development from scratch or wrappings". He further suggests that maintenance skills of developers are of predominant importance. We agree on this viewpoint especially in the context of open source

software systems, because there are typically only two types of documentation available: on the one hand, there is an initial vision document or feature lists on a very abstract level, and on the other hand, there is detailed source code documentation in form of code comments or API descriptions.

However, in cases when software development organizations want to adapt existing open source software systems to fit their own needs, this information is not sufficient because of the large gap between the two types of documentation. Most crucial for adaptation purposes is an architectural description which provides essential information about 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 [IE00]. Most open source systems however are lacking such an architectural description that could close this gap, and if there is architecture documentation, it is typically not maintained, outdated, or incomplete.

To successfully adapt open source software, it is necessary to derive architectural descriptions in order to understand them. Thus, the source code of the open source systems has to be analyzed and abstracted. Doing this manually is a time-consuming, complex work due to the high amount of data provided on a low level of abstraction. Architecture visualization and evaluation tools reverse engineer the source code and support the analysis tasks by software visualization. In particular, static architecture evaluation ([CKS05; Kn06; KS03]) of the software systems under investigation has been proven as a sound instrument to derive architectural abstractions and to ensure consistency between the implementation and the architecture. In [Kn06] we summarized our practical experiences of several case studies and identified a set of ten purposes and needs that drive subsequent architectural development. In this paper, we emphasize one of the purposes (namely "comprehension") and exemplify its importance in the analysis of open source software by analyzing the Apache Tomcat web server. We show further that with appropriate tool support it is possible to gain significant insights into the architectural concepts applied with limited effort. For the analysis of the medium-scale software system we spent less than a person-day effort.

Thus, we recommend the tool-supported static analysis and architecture evaluation to every software organization before deriving the reuse decision. Such architecture evaluation supports decision making among open-source alternatives, answering where to place the adaptations needed, what architectural concepts to consider and reasoning about the appropriateness of the candidates with respect to the own needs.

The remainder of this experience paper is structured as follows: Section 2 explains the basic principles of static architecture evaluation and presents an introduction to the SAVE (Software Architecture Visualization and Evaluation) developed at the Fraunhofer IESE. Then we describe our approach for the exploration of an existing software system with respect to reuse and apply it to conduct the Apache Tomcat case study, which is described in Section 3. The paper concludes with a summary in Section 4.

2 Static Architecture Evaluation

Static architecture evaluations analyze models of a software system without executing it. Such static analyses extract facts from existing artifacts, mainly the source code but consider also documentation, version control systems, etc. Extracting these artifacts is a reengineering activity. The output of static analyses comprises amongst others: static decomposition, hierarchies, static metric values, responsibilities, interfaces, naming conventions, dependencies, etc.

A special kind of static architecture evaluations is the analysis of the architecture conformance: a comparison of architectural models (the planned or intended architecture) against source code models (the actual or implemented architecture), as depicted in Figure 1 (based on the Reflexion model idea of [MNS01] and[KS03]). Thereby, each model consists of a set of (hierarchical) model elements and different types of dependencies (variable accesses, calls, etc.) between them. The comparison requires a mapping (typically a human-based task) between the two models and assigns evaluation types to each dependency:

- Convergence – dependency exists in both the architectural model and the source code model
- Divergence – dependency is only present in the source code model
- Absence – dependency is only present in the architectural model

The architects can interpret the results by the total numbers of convergences, divergences and absences. In some cases, it is necessary to calibrate the evaluations (i.e., refinement of the architectural or the source code model, or the mapping).

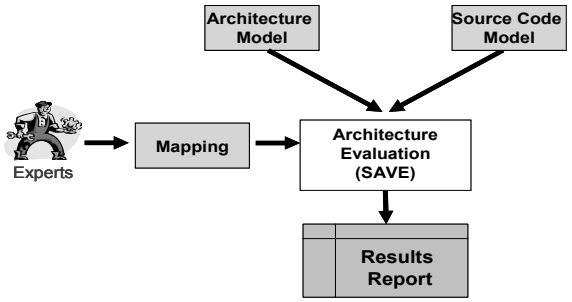

Figure 1: Principle of Static Architecture Evaluation with SAVE

Static architecture evaluations in general act as depicted in Figure 1, although not necessarily all of the three input artifacts have to be available. The outcomes are documented in a results report (graphical and textual), which can be processed further.

2.1 Purposes for Static Architecture Evaluation

As described above static architecture evaluations are a sound instrument in architecture development, the benefits were shown in several case studies, however there are certain questions that recurrently arise:

- Which models (architecture, components, code) at which granularity to use?
- How to interpret the high amount of data generated?
- How to integrate the results into further architecture development?
- What is the impact of the results on architectural decisions?
- How much effort to spend on iterations and refinement?

As described in [Kn06] our practical experiences gained in several case studies allowed us to systematize the applications of static architecture evaluations through a clustering into distinct purposes and needs that help answering the questions above, act as a perspective on the results, and drive further architecture development.

One of the ten purposes identified we named "**comprehension**" (please refer to [Kn06] for a definition of the other purposes) which is an assessment of program comprehension on a higher level of abstraction (i.e., an architectural level or a component level). Thereby mental models are reflected against the implementation and iteratively improved (e.g., as described in [Kn03]) until the mental model and the implementation converge. The mental model once harmonized with the source code serves then as basis for the architectural documentation. To avoid architecture degeneration over time architecture evaluations should be repeated at regular intervals, and, if necessary, the documentation should be updated accordingly.

To achieve the compliance of mental models and source code, the architects and developers have three possibilities for adjustment: either the architectural model, the mapping of or the source code itself can be changed:

- The architectural model captures the high-level view of the architects on the structural decomposition of the system. Changes to the architectural model may be necessary if the architects misconceive this high-level model (i.e., it is implemented differently), if architects have contradicting opinion on the system, or if architects refine their views on the systems due to refinements gained by the analyses results.
- The mapping is the "one to many" assignment of architectural elements to elements of the source code model. Changes or updates to the mapping if the mapping was incorrect and source code elements were accidentally assigned to the wrong architectural model elements, if architectural elements or source code elements are added or removed, if there were inconsistency in the mapping rules, or if architects had varying or contradicting mappings.
- The source code elements are a representation of the source code provided by fact extractors that parse the source code. Changes to the source code elements may be necessary if architects and developers agree on architectural violations and they are removed or refactored, if the system evolves, if the system was given in several variants (e.g., by preprocessor directives) or as instances of a product line and the next instance is

analyzed, or if the architects conduct “what if”-analyses and try out different structural decompositions.

To derive a first mental or architectural model there are two main strategies:

- **Bottom-up strategies** thereby aim at abstracting the models more and more. They start with an architectural model that is nearly a one to one copy of the detailed source code model. Then the architectural model is abstracted more and more by merging elements, by grouping them into clusters, or by defining containers.
- **Top-down strategies** refine abstract (domain) views until they conform to the details of the implementation. The very abstract architectural model comprising only a small set of architectural elements at the beginning is usually only partially mapped to source code model elements and then improved more and more by creating lower-level components and by adjusting the mapping.

However, the most promising approach in our experiences is a combination of both strategies. To handle the complexity of the results and to reduce the amount of data to deal with, it is an advantage to first focus only on a single part of the system (i.e., analyzing a subsystem or just a small set of components) under investigation. Once the compliance has been achieved for a single part, the analyses activities can then be spread out to other parts of the system.

Static architecture evaluations comparing the mental models and the implementation measure the distance between those two, i.e., the degree of compliance. We define architecture compliance as the measure to which degree the implemented architecture in the source code is realized compliant to the planned architecture as specified or intended (i.e., a compliance of 1.0 or 100% means that there are no architectural violations at all, 0.0 or 0% the opposite). Architecture compliance checking is the means to measure this.

2.2 The SAVE Tool

As the architecture is a crucial artifact in the lifecycle of a software system there must be effort spent to guarantee a high quality of that architecture. Therefore, it is necessary to evaluate the architecture continuously. This enables the early detection of shortcomings in the architecture. The earlier problems are detected the cheaper they can be solved. The idea behind the SAVE (Software Architecture Visualization and Evaluation) tool is to evaluate software architectures while they are constructed as well as after their construction. It was developed at the Fraunhofer IESE in Kaiserslautern. In order to allow for a good integration of the SAVE tool into the software development process it is designed as a plug-in for the Eclipse development environment. We created an Eclipse perspective for SAVE (see Figure 2) that encapsulates all its functionality.

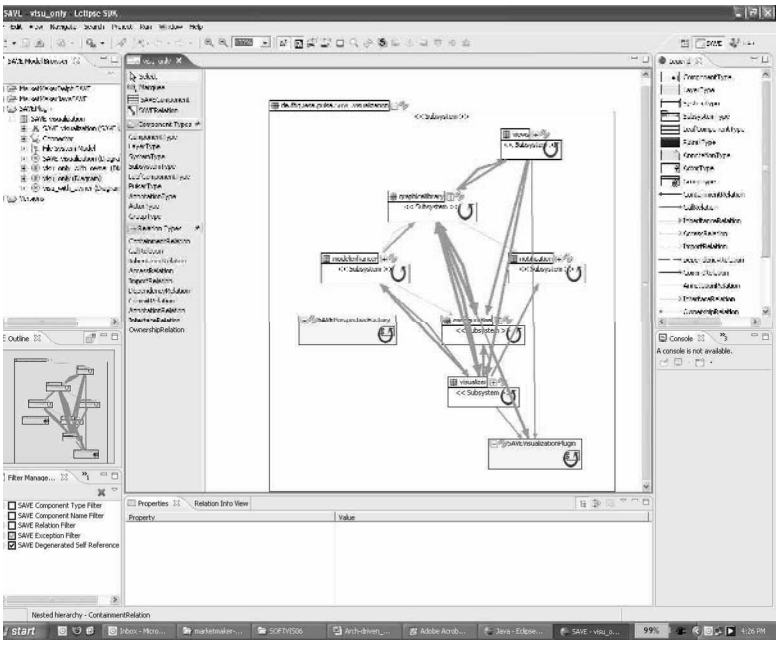

Figure 2: SAVE Overview

The static architecture of an existing software system can be extracted from the source code using the SAVE tool. It is possible to parse source code in the languages Java, C/C++, and Delphi and to represent the information extracted in a SAVE internal data format. The evaluation of the extracted architectures is supported in many ways. For example, there is functionality for comparing different states of an architecture over time. This is done by comparing the information about components and relations among different architecture states and figuring out where and what kind of changes happened. Therefore we developed different strategies, for example based on text matching of the component names. Further, an architecture can be evaluated against a reference architecture to detect where deviations from the intended architecture are. For this kind of evaluation the architecture as it is intended to be is modeled in the SAVE tool. Then a mapping to the architecture extracted from the code is done and finally the architectures are compared (see Figure 1). All results of these evaluations can be displayed in the visualization component of the SAVE tool. This allows for a fast comprehension of the problems detected. Apart from that the visualization can also be used to browse the architecture that was extracted from an existing software system. As highly abstracted views can be generated an experienced architect can quickly see where problems in the architecture could arise. Thus, the visualization of the SAVE tool offers many possibilities to support the evaluation of software architectures, among them architecture compliance checking as described in the previous section.

3 Case Study

In order to have a well-known, practical demonstrator (and not just a toy example) exemplifying the realization of different architectural concepts enabling architectural reasoning about pro and cons of architectural means, we have chosen the Apache Tomcat web server as a demonstrator. In this section, we first introduce Apache Tomcat and our approach; we then discuss the architectural analysis results in detail. All figures about Apache Tomcat in this section have been produced with the SAVE tool, the total effort for achieving the results was less than a person-day.

3.1 About Apache Tomcat

Apache Tomcat is a servlet container for the Java Servlet and Java Server Pages technologies. It is developed in an open and participatory environment and released under the Apache Software License. Apache Tomcat powers numerous large-scale web applications across a diverse range of industries and organizations [Ap05]. We used Apache Tomcat version 5.0.28 as the system to be analyzed in the case study. It is a system with about 4.5 MB of source code in 411 .java files in 42 packages.

3.2 Approach

To systematically address the analysis of the Apache Tomcat we instantiated two methods, Fraunhofer PuLSE™-DSSA (Product Line Software Engineering, DSSA stands for Domain-Specific Software Architectures) and Fraunhofer ADORE™ (Architecture- and Domain-Oriented Re-Engineering)[1] method.

PuLSE-DSSA results in (product line) architectures documented in a selection of architectural views. Since Greenfield scenarios are found only rarely in industrial contexts, PuLSE-DSSA is designed to smoothly integrate reverse engineering activities into the forward-engineering process of defining a (product line) architecture. ADORE™ (Architecture- and Domain-Oriented Reengineering) is a request-driven reengineering approach that evaluates existing components with respect to their adequacy and integration potential into another system (or product line). Our analyses activities were guided by the two methods, please refer to [Ba04] for detailed description.

3.3 Approach Instantiation

We analyzed the Apache Tomcat to have a medium-scale system that works well as a demonstrator for architectural concepts. Our goal was to exemplify the implemented architectural concepts and based on these results, to enable architectural reasoning about pro and cons of concepts chosen, and to discuss advantages and disadvantages of alternatives. To allow such reasoning certain aspects of the

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

component have to be lifted to a higher level of abstraction. The existing artifacts (e.g., source code, documentation) are the sources that can be exploited. The information extracted is stored in a repository, so that different views on the data can be constructed. The challenge thereby is to process the information and create meaningful views since the repository usually has much content and relevant information is often hidden in detailed low-level models.

In particular, the static analysis of the case study comprised the following main activities (each described in the next sections):

- Source code analysis
- Decomposition structure
- Communication protocols
- Layering
- Architecture evaluation
- System environment analysis

3.4 Source Code Analysis

The SAVE tool offers different strategies for abstracting the source code with its large number of details to a higher-level view. In order to save time we chose a coarse-grained abstraction strategy which constructs a high-level component for every java package of the system under investigation. However, despite of this abstraction, all the information about relationships is preserved. This means that a relation like a method call between classes is lifted to a high-level relation between the components there. Thus it is possible to recognize relations very simply by looking at the visualization. As even such a course-grained strategy can result in a large number of artifacts we included a reasonable number of possibilities to reduce the visual complexity. For example, components can be collapsed to hide all their inner details and relations can be aggregated for simply showing which components are interrelated.

We will demonstrate this source code analysis with a little example for better comprehension. In the following two very simple classes in two different packages can be seen. Figure 3 presents the result of parsing these two classes with the SAVE tool. There are two components shown that represent the packages in the source code. Three relations with different appearance symbolize the class import, the variable access and the method call from “Hello” to “World”.

```
package hello;

import world.World;

public class Hello {
    public void caller() {
        World world = new World();
        world.variable = "Hello World.";
        world.methodCall();
    }
}
```

```
package world;

public class World {
    public String variable;

    public void methodCall() {
        System.out.println(variable);
    }
}
```

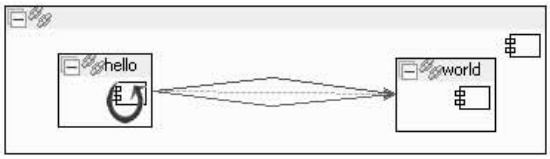

Figure 3: SAVE extraction and visualization example

The fact extraction of the internals of Apache Tomcat resulted in 9872 relations of the types package import, method call, variable access, inheritance, and interface implementation. Taking the relations to the libraries into account another 17823 relations were counted.

3.5 Decomposition Structure

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 4 presents a high level view of the system and the four main subsystems in a UML-based notation.

The `org.apache.catalina` subsystem represents the servlet container of Tomcat. It is responsible for the lifecycle of servlets and their correct delivery. Additionally the `org.apache.coyote` subsystem represents an HTTP connector, which enables catalina to act as a standalone web server. The main content of `org.apache.tomcat` are basic utilities like thread processing, buffer handling, and logging. Finally, `org.apache.naming` represents name server functionality.

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 example, there is the relation from the `org.apache.coyote` to the `org.apache.tomcat` component. This relation symbolizes 277 relations on the source code level, which were abstracted in this view for clear comprehension.

As one result we identified a cyclic dependency between the components `org.apache.catalina` and `org.apache.coyote`. Cyclic dependencies are often not desired and might cause problems in maintaining a system.

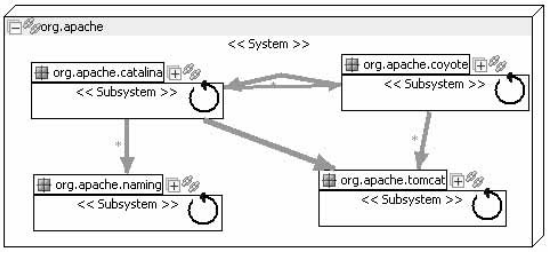

Figure 4: Apache Tomcat Overview

Further analyses of the system required navigating into collapsed components by expanding them within the visualization. Therefore, the user can explore 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 5 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).

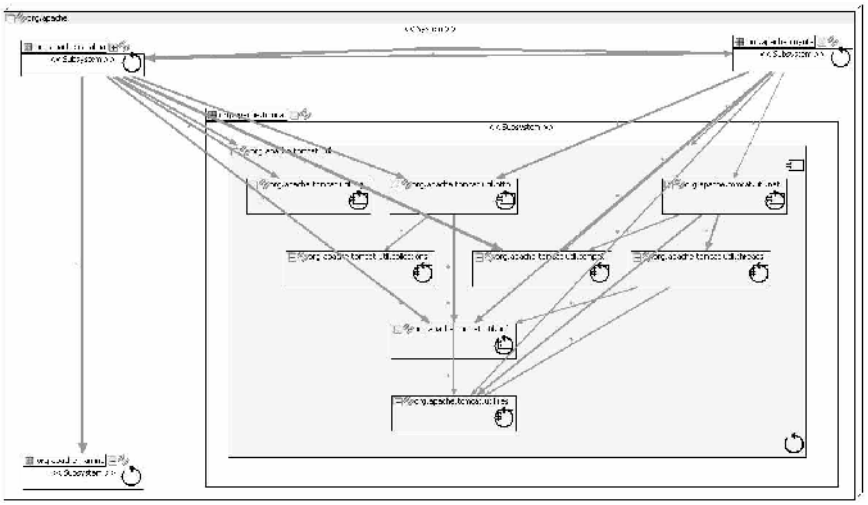

Figure 5: Zooming into a Subcomponent

3.6 Communication Protocols

The main application of Apache Tomcat is to provide web pages, thus communication protocols make up an important part of Tomcat, and several protocols are supported.

An interesting observation, which became an analysis task in a later experiment (see Section 3.10), 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 static analysis based on 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 6 presents an extraction of the parts involved in the protocol processing. Surveying a system with clear separation into packages and an appropriate naming it is possible to learn much about the semantics from the static structure. This example of the network protocols can give important hints if one is interested in adding new protocols by exploring similar properties.

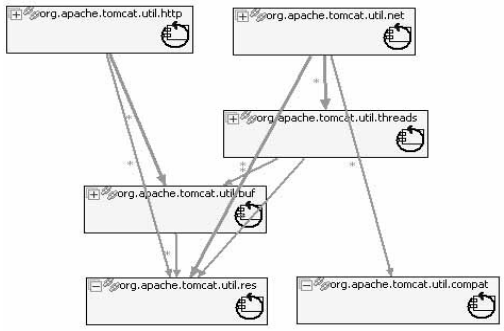

Figure 6: Extraction of the Communication Processing

3.7 Layering

Similar as the `org.apache.tomcat` we also investigated `org.apache.catalina`. We found that there are a large number of components in that package and thought about putting some order on it. Looking at the incoming and the outgoing relations of each package raised the idea to order the architecture according to the "Layering" architectural style. Therefore the ratio of incoming and outgoing relations was analyzed. 3 layers were thought to be adequate in order to keep clear assignments of components to layers. The layers are called ServiceLayer, ProcessingLayer, and LibraryLayer. In the SAVE tool there is functionality provided for computing metrics. For our layering approach the fan-in and fan-out metrics (whereas fan-in represents the number of incoming relations and fan-out represents the number of outgoing relations of a component) were used. The following formulas describe the assignment criteria for components to layers.

- Service Layer: $\#in / \#out \rightarrow 0$
- Processing Layer: $\#in / \#out \approx 1$
- Library Layer: $\#in / \#out \rightarrow \infty$

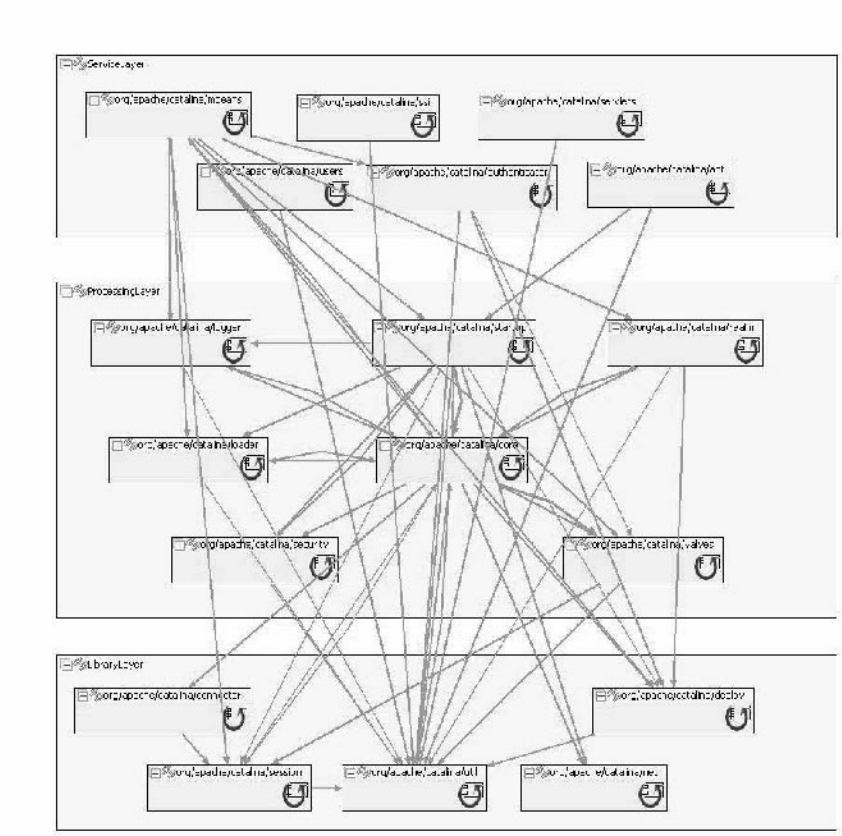

Figure 7: 3-Layer Architecture for catalina Package

3.8 Architecture Evaluation

In Section 2 we presented the idea of evaluating an architecture extracted from an existing software system against an intended architecture. Now we apply this for a part of the Tomcat architecture. We described our intended mental model of a 3-layer architecture for the `org.apache.catalina` subsystem before and showed how the components were assigned to the layers based on fan-in and fan-out metrics. In order to evaluate how close our component assignment is to a strictly layered architecture, we modeled a simple, strict 3-layer architecture as shown in Figure 8 (we only allow access from a layer to the layer directly beyond it). Then the mapping of the components to the layers is done and the SAVE tool automatically computes the evaluation result. In Figure 9 this result can be seen. The green checkmarks show that the relations are in the architecture as they were intended; the yellow exclamation marks symbolize the divergences (i.e. there is a relation in the source code that was

not intended). However, what is not obvious in this figure is the fact that the relations that symbolize convergences stand for 948 relations in the source code, while the relations that symbolize divergences stand for 366 relations in the source code (The relations are aggregated in the figure for more clarity). Investigating these numbers in more detail reveals that by far the most diverging relations go from the Service Layer to the Library Layer. This leads to the assumption that we should not propose a strict layered architecture. Then calls to deeper layers than the one directly below are allowed as well. Changing the intended architecture model and executing the evaluation again leads to the following result: 1289 convergences and only 25 divergences. Most of these divergences are related to the class "StandardContext". We exactly know the locations that cause the divergences and the SAVE tool allows to navigate to the respective source code fragments. For more information a detailed code analysis is necessary. In short, we found a compliance degree of around 98% and thus our intended model can be seen as very close to the implemented architecture.

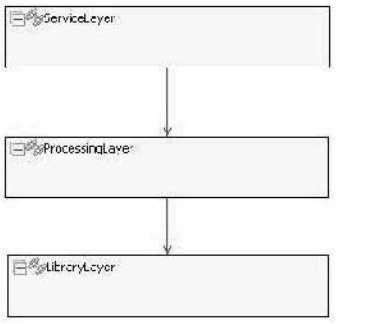

Figure 8: Intended Architecture (initial mental model) for the catalina subsystem

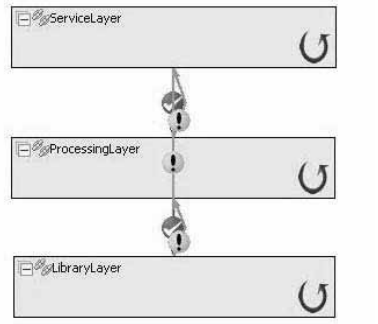

Figure 9: Result of the Architecture Evaluation

The evaluation results make us confident that our ordering into 3 layers is feasible and can enhance the comprehensibility of the Tomcat architecture. Thinking about

extensions of the `org.apache.catalina` subsystem may be eased as the layers propose where to add new components.

3.9 System Environment Analysis

Another analysis we did on the Tomcat system was to extract the overall static architecture as described before including the relations to components outside the Tomcat itself. This means that all the libraries that are included are analyzed, too. With respect to the adaptation of open source software this offers a fast overview of the libraries that are involved in the parts of the system that will be adapted. In Figure 10 a high-level abstraction of this can be seen. It can be, however, zoomed down to the package level of the libraries as shown before. Considering reuse of a system it is possible to decide against or for a system according to the libraries that are used and might be seen as unreliable.

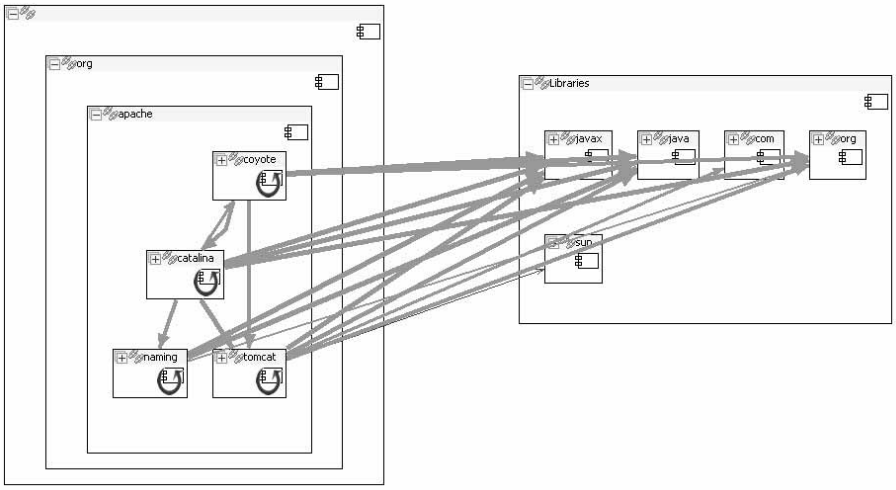

Figure 10: Tomcat analyzed with libraries involved

3.10 Architecture Analysis Experiment

In [Na05] an experiment was conducted for the validation of the SAVE visualization itself. The main goal of the experiment was to investigate the adequacy of the graphical elements in SAVE for the representation of architectural concepts. 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. As architectures cannot be demonstrated and analyzed in “Hello World” examples we searched for a system of appropriate size. Due to the knowledge gained in the architecture analysis described above we chose Apache Tomcat. With the help of the

results produced with the SAVE tool, we were able to design appropriate experimental tasks.

3.11 Summary

In the case study we demonstrated that deep insight in the structure of a software system can be gained efficiently when carrying out the analysis activities tool-supported and partially automated. With only little effort and in short time we achieved a degree of knowledge for which we would otherwise have to invest much more effort. Thus, SAVE facilitates the exploration of the Apache Tomcat. It also can be used to analyze more than one software system or different variants of the same system before deciding upon to use or adapt one of them. We think another benefit of SAVE is its possibility to create different views addressing different aspects of the same system integrated in one tool environment, linking the source code to the graphical representation of the Tomcat architecture.

Comparing effort and result we can say that the usage of the SAVE tool for the purposes we presented in this paper pays off. The one person-day effort we spent for the static architecture analysis activities was well-invested yielding in insights that would have been very difficult to achieve with just the given documentation and the source code itself.

4 Conclusion

The adaptation of open source systems is one promising strategy for today's software development organizations in order to address the market requirements. Reuse through adaptation offers the potential of profiting from existing, field-tested, valuable solutions. However, the main challenge is to get familiar with the open source systems which are considered as reuse candidates. Typically, such systems lack an architecture description, and thus, it is difficult to reason about the adequacy of several candidates since the internal, major architecture concepts are unknown. Thus, before reusing a candidate it has to be understood by the architects who derive the reuse decision.

Our analysis of the Apache Tomcat web server demonstrated that it is possible to gain significant insights on an architectural level in relatively short time (e.g., layering, cyclic dependencies, communication protocols, system environment). We now know about major architecture principles applied although they might be only implicitly intended by the developers. Static architecture analyses facilitate the architecture comprehension of a system. Their results guide then detailed program comprehension on the source code level, which is still necessary when eventually reusing and adapting an open source reuse candidate. Static architecture analyses enable and support such adaptations by providing a broad view on the complete system (i.e., the "big" picture) and they can identify adaptations hooks where best to modify the code for the own purposes.

The SAVE tool, an Eclipse plug-in developed at Fraunhofer IESE, supports static architecture evaluation. We were able to conduct the case study in a fast and efficient way requiring a total effort of a person-day. SAVE primarily supports architecture

comprehension tasks and provides by its nature as an Eclipse plug-in an integrated link to the source code and other Eclipse plug-ins.

We plan to monitor the evolution of Apache Tomcat on an architectural level and to learn about the impact of architectural changes. Tracking of the evolution is an important task once the adaptation of an open source system has been made: when switching to a new release of the open source system, risk management has to carefully assess the impact of changes in adapted code and to thoroughly analyze the potential consequences to the own system.

5 References

[Ap05] Apache. (2005). *The Jakarta Site - Apache Tomcat*. http://tomcat.apache.org/index.html

[Ba04] Bayer, J., Forster, T., Ganesan, D., Girard, J.-F., John, I., Knodel, J., Kolb, R., & Muthig, D. (2004). *Definition of Reference Architectures based on Existing Systems* IESE-Report 034.04/E.

[CKS05] Christl, A., Koschke, R., & Storey, M.-A. (2005). *Equipping the Reflexion Method with Automated Clustering.* Working Conference on Reverse Engineering, Pittsburgh, USA.

[IE00] IEEE. (2000). *ANSI/IEEE Std. 1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems*.

[Kn03] Knodel, J. (2003). *Reconstruction of Architectural Views by Design Hypothesis* Softwaretechnik-Trends, 2003.

[Kn06] Knodel, J., Lindvall, M., Muthig, D., & Naab, M. (2006). *Static Evaluation of Software Architectures.* 10th European Conference on Software Maintenance and Reengineering, Bari, Italy.

[KS03] Koschke, R., & Simon, D. (2003). Hierarchical Reflexion Models. *Proc. of the Working Conference on Reverse Engineering, IEEE*.

[MNS01] Murphy, G. C., Notkin, D., & Sullivan, K. J. (2001). Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE Trans. Softw. Eng., 27*(4), 364-380.

[Na05] Naab, M. (2005). *Evaluation of Graphical Elements and their Adequacy for the Visualization of Software Architectures* IESE-Report 078.05/E. Kaiserslautern.

[Pi04] Pizka, M. (2004). *Adaptation of Large-Scale Open Source Software.* 8th European Conference on Software Maintenance and Reengineering, Tampere, Finland.