Fraunhofer Institut
Experimentelles
Software Engineering

The SAVE Plug-in - Internal Data Model and Architecture Evaluation Functionality

Authors:
Dominik Rost
Thomas Forster
Jens Knodel

IESE-Report No. 063.06/E
Version 1.0
June 8, 2006

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 paper presents a solution for the evaluation of a model generated from an existing system against a planned architecture, to identify potentially occurring differences between the architecture and the implementation as soon as possible, in form of a plug-in for the Eclipse platform with the name of *SAVE - Software Architecture Visualization and Evaluation.* Besides the SAVE Core Model with which models on a high level of abstraction as well as models close to the implementation of a system can be built the evaluation process is explained in detail. Further more some special aspects of the plug-ins implementation which includes the plug-in structure and fact extraction with the Java Development Tools provided by the Eclipse SDK are explained. In addition a small case study is given to demonstrate the evaluation of a software architecture.

Keywords: Architecture Evaluation, Eclipse, Software Architecture, Static Analysis, Visualization, PuLSE, SAVE

Table of Contents

1 Introduction

The creation of software architectures and their visualization, in different ways and from different perspectives are important means for developing complex software systems and essential elements of software engineering processes. When applying such means one can gain some significant advantages. Especially the planning on a high level of abstraction and thus, the disregarding of (at this moment) irrelevant details allows a complete overview over the system and therefore the identification of possible entities and relations among them.

The following sections will present the common understanding of software architecture followed by the mentioned risks and the motivation and the presentation of the problem as a consequence of these risks.

1.1 Software Architectures

There is a variety of definitions for software architecture or architecture in general .In [Sof06] the following definition of [BCK03] can be found, which may be one of the most common accepted definitions of software architecture.

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*
>
> *"Externally visible" properties refers to those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on.*

[BCK03] elucidates the implications of this definition in the following way:

- The architecture embodies information about how the elements relate to each other. This means that architecture specifically omits certain information about elements that does not pertain to their interaction. Therefore software architecture is an abstraction and does not represent internal details.
- Systems can and do comprise more than one structure and no one structure holds the irrefutable claim to being the architecture.
- Every software system has an architecture because every system can be shown to be composed of elements and relations among them even if the system is monolithic.

- The behavior of each element is part of the architecture insofar as that behavior can be observed or discerned from the point of view of another element.
- The definition is indifferent as to whether the architecture for a system is a good one or a bad one

The definition imparts a good idea of software architecture. It should be pointed out that there are also different perspectives to look at a software architecture, i.e. *views,* depending on what characteristics are of interest. Examples of such views are the *logical view, physical view* or the *conceptual view*, et cetera.

1.2 Motivation

Software systems continuously reach new dimensions of size and new levels of complexity. By using visualizations of software architectures on a high level of abstraction and derived diagrams of single systems containing more details it is possible to develop good and consistent systems. But often and increasingly over time discrepancies emerge between the planned architecture or structure of subsystems and the real implementation in the form of source code. The later such discrepancies are detected the more time- and cost-consuming the necessary changes will be to make the system consistent to the specified architecture.

The question is how such mistakes in the implementation can be detected as early as possible. The aim of reengineering solutions is often the generation of models from existing systems. These models are useful in respect to representing the system most accurately. But a model presenting an architecture which was derived from a planning process often has a typically different abstraction level than a generated model, which makes it difficult to compare them. Therefore mistakes or implementations conflicting with the planned architecture are hard to detect.

This paper will present a possible solution for the mentioned problems. On the basis of the idea of the *reflexion models* we developed a tool which allows the detection of implementations that conflict with a planned architecture which allows correcting these mistakes as early as possible.

2 Related Work

2.1 Evaluation of Software Architectures with Eclipse

The solution presented in this paper is a further development of Paul Miodonski's concept of the SAVE-plug-in introduced in *Evaluation of Software Architectures with Eclipse.* However in some fields the two versions are basically different. The goal of the redevelopment was to re-implement all existing features but with attention to better flexibility and extensibility.

2.2 Evaluation of Graphical Elements and their Adequacy for the Visualization of Software Architectures

In *Evaluation of Graphical Elements and their Adequacy for the Visualization of Software Architectures* Matthias Naab introduces a concept for the visualization of software architectures. During his work this concept was implemented as an independent plug-in and integrated into the SAVE-plug-in. This extension serves therefore as the component responsible for displaying all results produced by the SAVE-plug-in.

2.3 Static Evaluation of Software Architectures

In *Static Evaluation of Software Architectures* [KLMN06] the authors summarizes there experiences with several industrial and academic case studies where the architecture of existing system has been assessed. The tool used to conduct the case studies was the SAVE plug-in using the functionality described in the next sections.

3 Background

The following sections will describe some issues that form the basis of the development of the SAVE-plug-in. This is on the one hand the idea of the *reflexion models* which are the theoretical basis of the evaluation of software architectures and on the other the Eclipse project which provides the technical framework for the development of the plug-in.

3.1 Reflexion Models

The Evaluation of software architectures is based on the idea of the *reflexion models*, introduced by Murphy, Notkin und Sullivan in [MNS01]. Reflexion Models offer the possibility to evaluate models on a high level of abstraction (*high level model*) as they are often used by software architects in form of box-and-arrow-diagrams against the source code of a system and to analyze it in a way that makes commonalities and differences easily recognizable.

The result of such an evaluation is a reflexion model and is achieved in the following way: We assume that there exists a high level model, i.e. a planned architecture on a high level of abstraction and that we abstract from the source code of a system to gain such a model. A map is built to get a relation between the components of the high level model and the ones of the source code model. Now a reflexion model can be computed.

To all relations one of the following types is assigned:

- Convergence: The relation exists in the high level model as well as in the source code model.
- Divergence: The relation exists in the source code model but not in the high level model
- Absence: The relation was planned in the high level model but does not exist in the source code model

The following figure based on [Mio04] illustrates the issue:

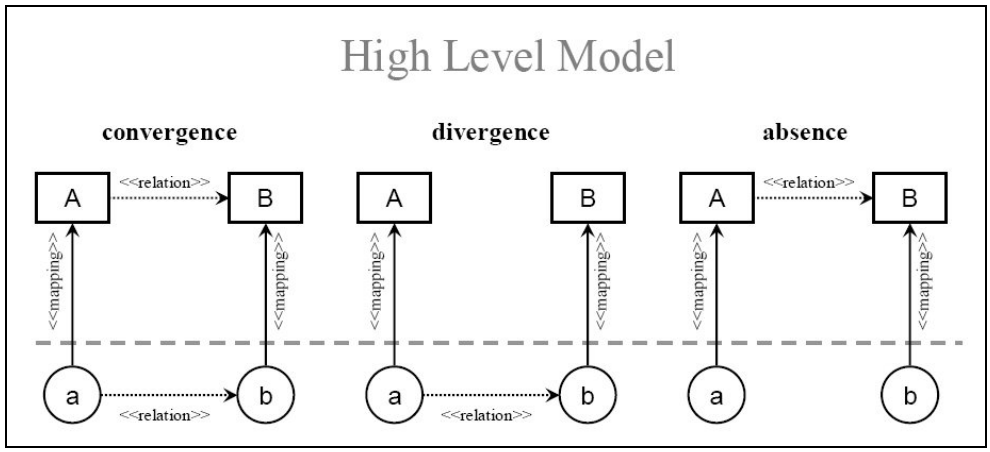

Figure 3-1 Reflexion Model Evaluation Types

[KS03] additionally presents an extension to the reflexion models, with which hierarchical Reflexion models can be computed.

3.2 Eclipse

Eclipse makes up the basis for the development of the tool. The following sections differentiate between the platform, the software development kit and the eclipse modeling framework.

3.2.1 Eclipse Platform

> *"The Eclipse Platform is an open extensible IDE for anything and yet nothing in particular."* [Ecl02]

This definition for the question of what Eclipse is, is very often to be found in [Ecl02] as well as in many documents and presentations addressing topics that are related to the Eclipse platform. The platform is very generic, i.e. it can deal with a great variety of different files and data, but only in a generic manner and not specialized for a particular data type. The main goal is therefore not to provide as much functionality by the platform itself but to make it as extensible as possible to exactly meet the needs of a developer.

The architecture of the platform reflects this approach (from [Ecl03]).

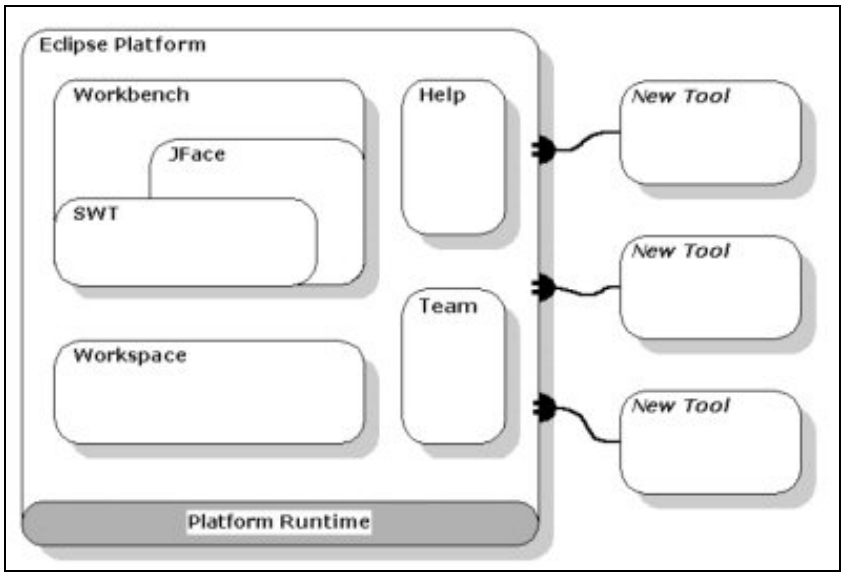

Figure 3-2 The Eclipse Platform Architecture

The platform provides the necessary services and frameworks to build new features and integrate them into it. This is done in the form of plug-ins which are embedded by plugging them into defined extension points. The platform runtime is the only part that is no plug-in but administrates the integrated plug-ins.

3.2.2 Eclipse SDK

The *Eclipse Software Development Kit* extends the platform with two plug-ins, the *Java Development Tools (JDT)* and the *Plug-in Development Environment (PDE)*. The JDT provide the complete development environment for the development of Java-projects. The PDE adds the functionality of building custom plug-ins for the Eclipse platform. This makes the platform extendable for the needs of the developer.

3.2.3 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is an Eclipse-plug-in which belongs to the *Eclipse Tools Project* and consists of a modeling framework and a code generator. Models are specified and stored in *XML Metdata Interchange (XMI)* data. Besides that models can be specified using *Annotated Java, XML documents* or with the help of tools like *Rational Rose* and then be imported to EMF.

From such specifications EMF can generate Java Classes for the model, adapter classes and basic editors. All models generated by EMF are fully compatible and

have the common basis for the realization of interoperability. Furthermore notification algorithms for model changes, persistent storing of models and a mighty reflective API are significantly important.

4 SAVE Core Model

The Core Model contains the main data structures for building the SAVEModel representing the architecture of the analyzed software system. Logically it can be split into three different entities that carry different types of information:

- *SAVEComponentModel*
- *FSModel*
- *SAVEFSConnector*

The following sections will describe each single part in detail, i.e. the contained components and the relations among them, as well as the collaboration between those three parts.

4.1 SAVEComponentModel

The SAVEComponentModel contains the data structures necessary to build the high level model, i.e. the model representing the software architecture on a high abstraction level. The following figure is an UML diagram showing those data structures as well as how they are related to each other.

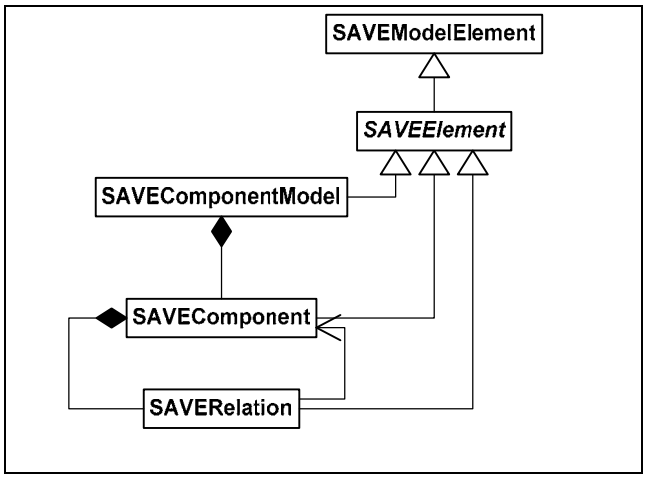

Figure 4-1 SAVEComponentModel UML Diagram

The main data structure of the SAVEComponentModel is the *SAVEComponent*, which represents entities of the software system on a high abstraction level. It contains an attribute that specifies its type to be capable of representing vari-

ous types of software architecture components, depending on the abstraction process from which it was built.

SAVERelations are connectors between SAVEComponents representing some kind of relation between entities of the software system. As well as SAVEComponents SAVERelations contain an attribute that specifies its type, reflecting the kind of relation of the software system that it abstracts. SAVERelations are contained in SAVEComponents or, more precisely in the SAVEComponent that is the origin of the relation. The target component is only referenced in the relation. All SAVEComponents know what relations they contain and from what relations they are referenced.

The *SAVEComponentModel* is a container for SAVEComponents and SAVERelations and is with their entirety the abstraction of the analyzed software system or of a part of it.

Obviously all the described data structures extend the abstract *SAVEElement*, which indicates that these are members of the model that abstracts the software system in contrast to the ones contained in the *FSModel*. The SAVEElement extends *SAVEModelElement* which all members of the core model extend.

4.2 FSModel

The FSModel is an abbreviation for "File System Model" and a representation of the analyzed software system on an abstraction level very close to the operating system. It also holds information at a higher level of detail i.e. entities within source code files like attributes and methods. Therefore the information it holds bears some resemblance to the information that can be gained from a file system or package browser like for example the ones in the Eclipse SDK. The idea is however to only show information that is relevant to the analysis so that the number of shown entities and relations is reduced and therefore information will become better accessible.

The following UML diagram shows the structure of the FSModel.

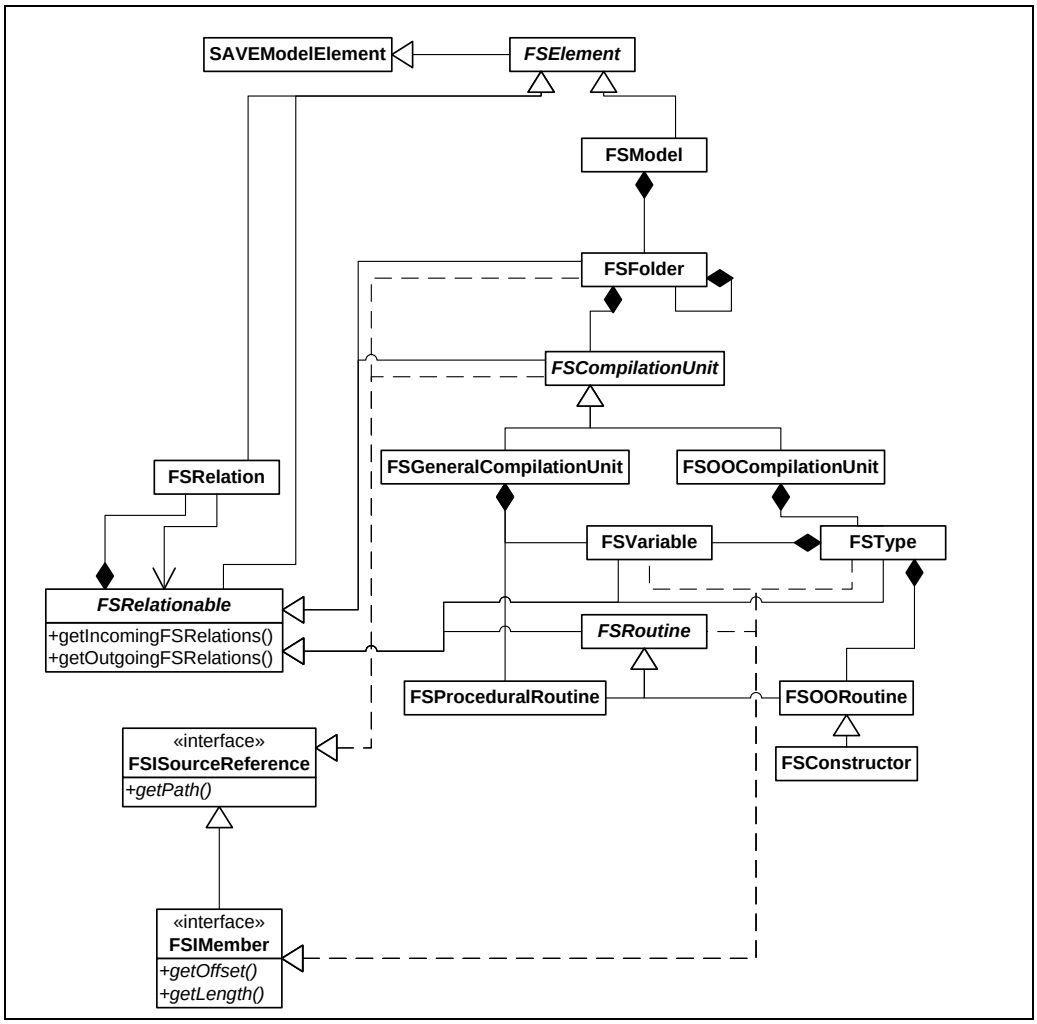

Figure 4-2 FSModel UML diagram

The structure on the top level is very similar to one of the SAVEComponent-Model. There is also a container for the model components that represents the whole model, which is here the *FSModel*. It also extends a component that indicates that it is a member of the FSModel, namely the *FSElement*, which in turn extends SAVEModelElement, like the SAVEComponentModel component in the SAVEComponentModel does.

The components lying under the FSModel component are used to build a representation of all relevant parts of the analyzed software system. Particularly important about it is the differentiation between object oriented and other, mostly procedural programming languages, realized by the *FSGeneralCompilationUnit* and the *FSOOCompilationUnit* components on the one hand and by the *FSProceduralRoutine* and the *FSOORoutine* and *FSConstructor* components

on the other. The benefit of this separation of programming language concepts is an increase of flexibility and extendibility regarding the kinds of software systems that can be processed. The elements in detail are:

- FSFolder: represents a directory, which can in the case of an object oriented language such as Java also be interpreted as a package. It can contain other FSFolders or FSCompilationUnits.
- FSCompilationUnit: represents a compilation unit such as a .java source file in Java context. FSOOCompilationUnit and FSGeneralCompilationUnit extend this for the reason explained above.
- FSType: represents an object oriented type such as a class or an interface and can contain FSVariables or FSOORoutines.
- FSOORoutine: represents a function, procedure or method in an object oriented context and extends therefore FSRoutine. It is contained in an FSOOCompilationUnit.
- FSConstructor: is a special FSOORoutine and represents a constructor of a class.
- FSProceduralRoutine: represents a function or procedure of a not object oriented programming language and extends therefore FSRoutine. It is contained in an FSGeneralCompilationUnit.
- FSVariable: represents a variable or member of the context being specified by the element that contains the FSVariable, which can either be an FSGeneralCompilationUnit or an FSType.

To reflect relations between entities in the software system like calls, imports or accesses there has to be a connection between FSModel elements which is realized by the *FSRelation* component. Like the SAVERelation it is capable of representing various types of relations and is contained in the element that is the source of the relation. The ability to build such a connection is inherited by FSRelationable.

The two remaining interfaces *FSISourceReference* and *FSIMember* extend implementing elements with information of the path, the offset and the length of the file system element they represent.

4.3 SAVEFSConnector

The *SAVEFSConnector* bridges the gap between the two models described in the previous sections by building references from every SAVEElement to n FSElements. Thus the information that can be extracted from this mapping is which file system model elements are represented by what element in the high level model.

The structure of the SAVEFSConnector is shown in the following UML diagram.

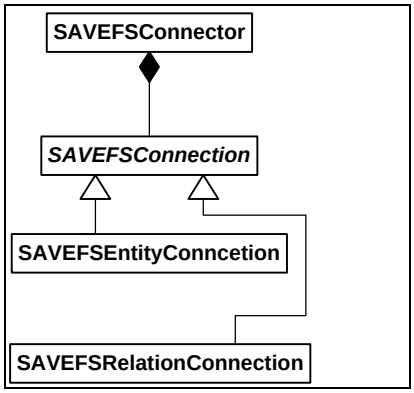

Figure 4-3 SAVEFSConnector UML diagram

The SAVEFSConnector holds SAVEFSConnections which build the mapping between SAVEElement and FSElements. The *SAVEFSEntityConnection* connects entities, the *SAVEFSRelationConnection* relations to each other.

4.4 SAVE Core Model

The three models described in the previous sections together form the SAVE core model and are thus the basis for all processes in analyzing a software system with SAVE. The following UML diagram shows, how those three parts are linked and how collaboration works between them.

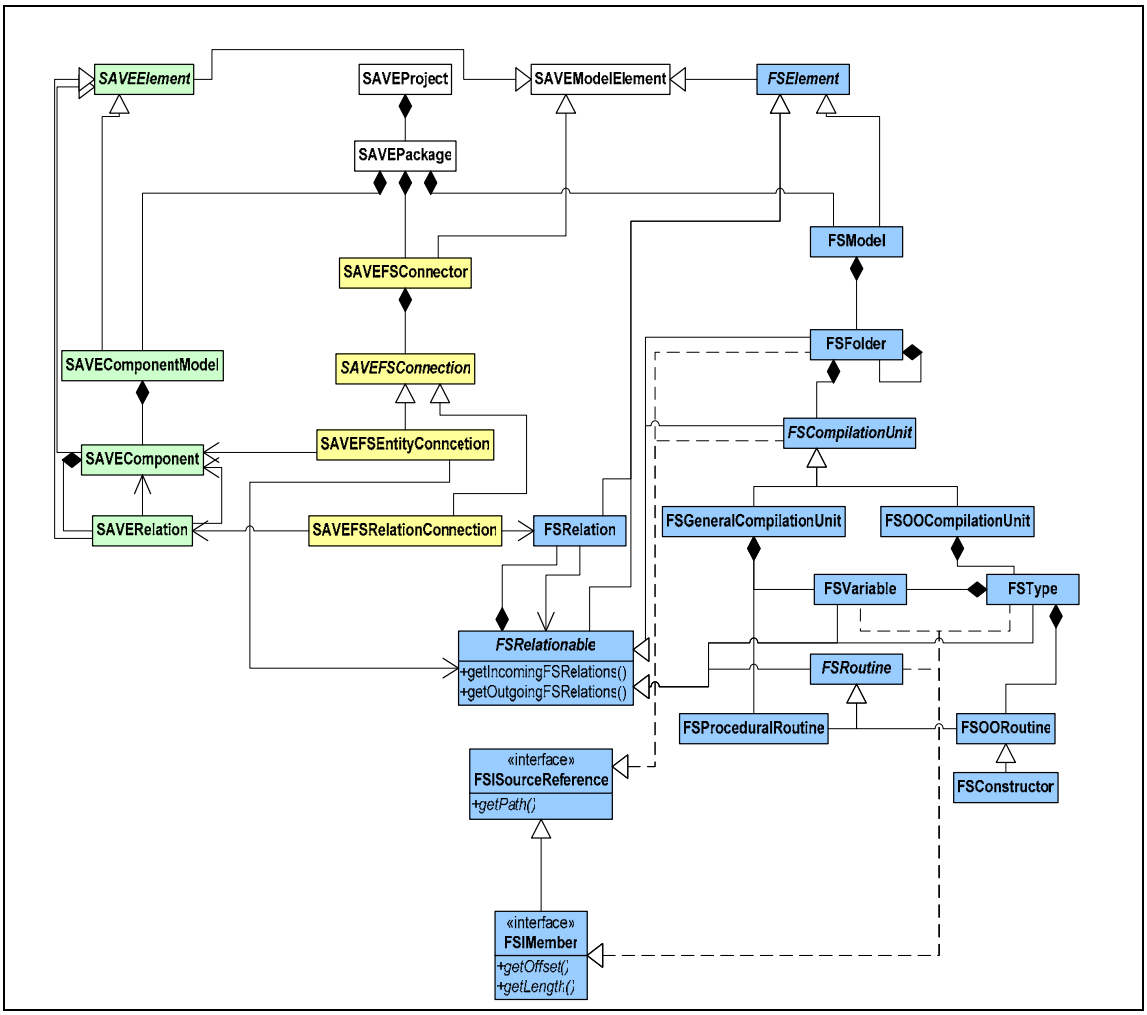

Figure 4-4 SAVE Core Model UML diagram

For the reason of better differentiability some colors have been added to the components depending to what model they belong. Members of the SAVE-ComponentModel are green, the ones of the SAVEFSConnector yellow and FSModel elements are blue. Apart from these there are some uncolored components and new connections which illustrate the collaboration between the single models.

The container for exactly one of each of these models is the *SAVEPackage* which in turn is contained in the *SAVEProject*. Since it should be possible to analyze several systems or several parts of a system the SAVEProject can contain multiple SAVEPackages, one for each analyzed system.

As already mentioned the SAVEFSConnector bridges the gap between the independent models by building connections between elements. The arrows going from the two types of SAVEFSConnections to components and relations of both models illustrate this. More precisely the SAVEFSConnection type holds references of the elements between which a mapping is to be built. The SAVEFSRelationConnection maps FSRelations to SAVERelations, the SAVEFSEntityConnection FSRelationables to SAVEComponents.

5 Evaluation

The Evaluation is based on the idea of the reflexion models described in chapter 3.1. To be able to evaluate an existing software system against a planned architecture the system has to be abstracted and a mapping has to be built. The abstraction process as well as the building of the mapping will be explained in the following sections.

5.1 Abstraction

Basically before an evaluation there exist a software system and a planned architecture which should be evaluated against each other. But since two models on the same level of abstraction are a precondition to do an evaluation, a model has to be abstracted from the software system, i.e. the system has to be lifted. The model generated in this way is the *Source Code Model (SCM)*.

Different strategies for this lifting process are imaginable. Packages or directories respectively or compilation units are possible elements from which entities of the SCM could be generated. From relations between or below packages or compilation units are then abstracted to relations between the entities. The following figure illustrates the issue.

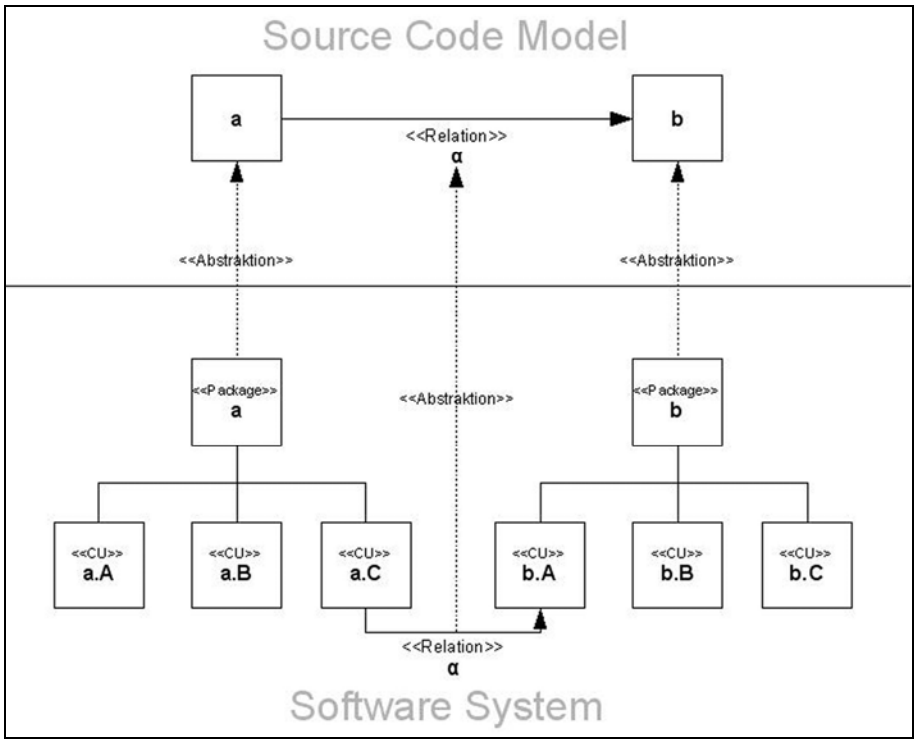

Figure 5-1 Model Abstraktion

Based on the SAVE-plug-in depending on the chosen strategy packages or compilation units are abstracted to SAVEComponents and relations to SAVERelations. The resulting SAVEComponentModel can be evaluated against the High Level Model, which in this case is also a SAVEComponentModel.

5.2 Evaluation Mapping

When the Source Code Model and the High Level Model are available a mapping has to be built. Since the Source Code Model is generated from an existing system and the High Level Model represents a planned architecture which is far less detailed, the Source Code Model tends to consist of more components than the High Level Model. Therefore a mapping has to be built as an 1:n-relation which allows a mapping of many SCM-components to one HLM-component.

The *SAVEEvaluationMapping* generates such a mapping by building references between SAVEComponents of the HLM to ones of the SCM.

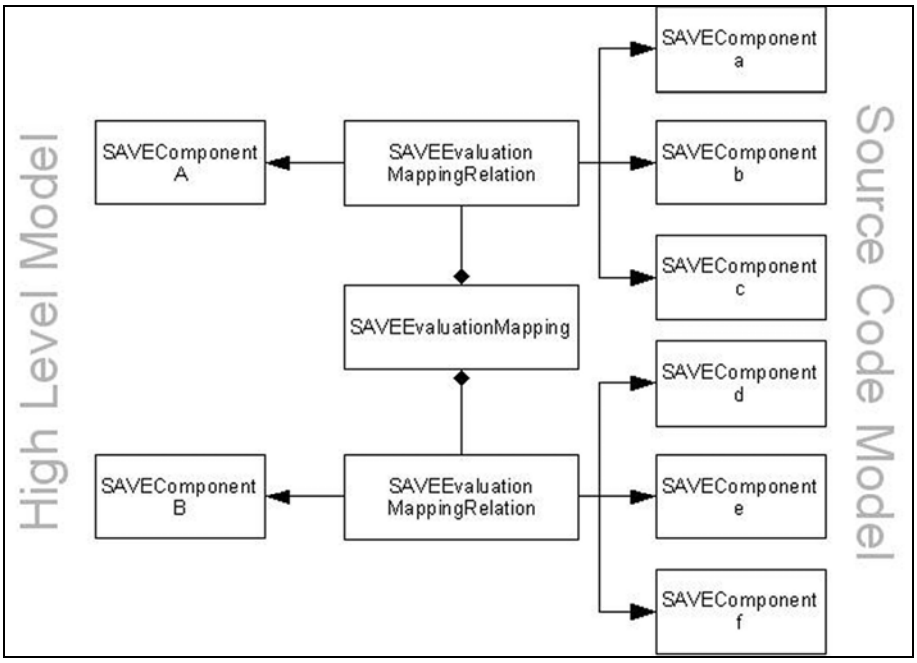

Figure 5-2 SAVEEvaluationMapping

The evaluation process runs in two phases and with it in two directions: from the high level model to the source code model and vice-versa. During the first phase all relations of all SAVEComponents are checked. If a corresponding relation for a certain relation is found in the source code model the relation is marked with the evaluation type *convergence*, otherwise with *absence*. During the second phase relations of the type *convergence* are identified which means that no corresponding relation is found in the high level model.

All SAVERelations contain the attribute *SAVEEvaluationType* which can be set during the process and can be used for the visualization of the model.

6 Implementation

The following sections cover some specific aspects of the implementation of the SAVE plug-in. Besides the structure of the plug-in and the implementation of the fact extraction an example is given which demonstrates the use of the plug-in to evaluate a software architecture.

6.1 Plug-in structure

The SAVE-tool is a plug-in for the Eclipse platform. Because of its size and complexity it is reasonable to unitize the plug-in, i.e. in this case to separate it in several different plug-ins On top level three different systems can be differentiated. *de.fhg.iese.pulse.common* is the basis for reuse which means that it contains plug-ins that can be used cross-project, which complies with the software product line approach used in this project. *de.fhg.iese.pulse.fe* combines plug-ins that are used for fact extraction, or more precisely, for every supported language in one plug-in. The third system is *de.fhg.iese.pulse.SAVE* which contains the core model and the core features of the SAVE-plug-in.

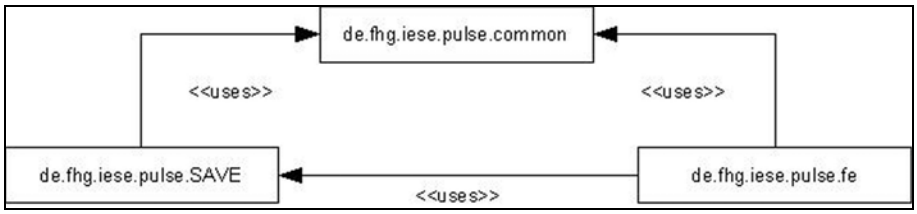

Figure 6-1 SAVE plug-in structure

6.2 Fact Extraction with the JDT

The *Java Development Tools* are part of the Eclipse SDK and contain the subprojects *JDT APT*[1]*, JDT Core, JDT Debug* and *JDT UI.* The JDT Core project provides a tool which makes the extraction of relevant facts from java projects very easy.

The ASTParser[2] builds up a syntax tree which means that for every relevant fact a node is generated which has exactly one parent and can have many children. With an *ASTVisitor* this tree can be traversed recursively. The following source

[1] Annotation Processing Tool

[2] AST = Abstract Syntax Tree

code listing illustrates how the classes are to be used to create a syntax tree and traverse it.

```
protected void parseFile(ICompilationUnit unit) throws Java-
ModelException {
 ASTParser parser = ASTParser.newParser(AST.JLS3); // 1.
 parser.setResolveBindings(true);
 parser.setSource(unit); // 2.
 CompilationUnit rootCU = (CompilationUnit)
 parser.createAST(null); // 3.
 if (rootCU != null) {
  rootCU.accept(new JavaASTVisitor(unit, saveModel, fsModel,
connector)); // 4.
 }
}
```

This is what is done during the execution of the source snippet:

1. A new parser object is generated. The parameter is an integer constant and specifies the API level.
2. A source is specified which is in this case a compilation unit, i.e. a .java file.
3. The parser generates the syntax tree.
4. The recursive traversing of the tree is started. Therefore a JavaASTVisitor object is generated which extends the JDT class ASTVisitor. It contains *visit*- and *endVisit*-methods overloaded for every relevant fact. These methods contain the source code to be executed when a certain type of node is reached.

The following methods contain implementations because the corresponding nodes represent relevant facts in our case:

- visit(ClassInstanceCreation node)
- visit(ImportDeclaration node)
- visit(MethodInvocation node)
- visit(SuperMethodInvocation node)
- visit(TypeDeclaration node)
- visit(SingleVariableDeclaration node)
- visit(VariableDeclarationFragment node)

6.3 Case Study

The following example demonstrates the evaluation of a software architecture. The analyzed software system is a small one and of low complexity to make the process of the evaluation and the results better understandable.

The system consists of three packages with one class each. All of these classes contain a method *call* to generate relations between the components. The source code of the three classes is as follows:

```
package scm1;
public class SCM1 {
  public static void call() {
    scm2.SCM2.call();

}

package scm2;
public class SCM2 {
  public static void call(){
    scm3.SCM3.call();
  }
}

package scm3;
public class SCM3 {
  public static void call(){}
}
```

The relations are calls between the methods. That means there is a call from the call-method of class *SCM1* to the one *SCM2* and from *SCM2* to *SCM3*.

The first step is the fact extraction from the software systems source to generate the source code model which generates the following model whereas a strategy to build components from packages was used.

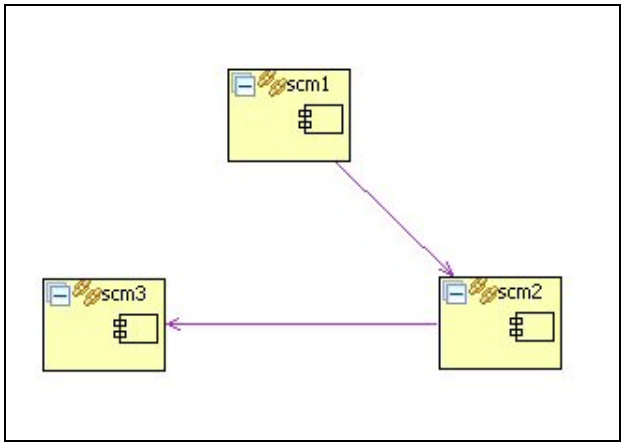

Figure 6-2 Case Study: Source Code Model

The model shows three components which represent the packages as well as two purple connectors that represent the calls.

The high level model against which the source code model will be evaluated also consists of three components and two relations but in this case the relations are from *hlm1* to *hlm2* and from *hlm3* to *hlm1*.

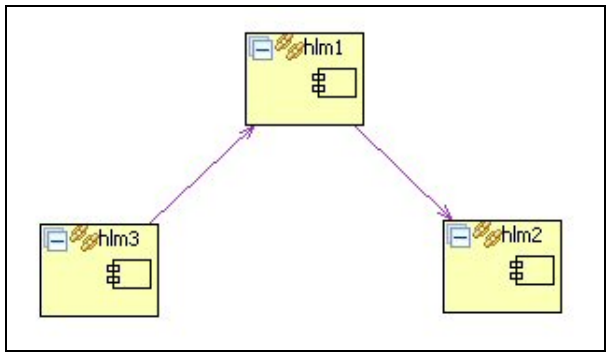

Figure 6-3 Case Study: High Level Model

Now a mapping between the components of the SCM and the HLM has to be created. The following figure shows the dialog to create a SAVEEvaluation-Mapping.

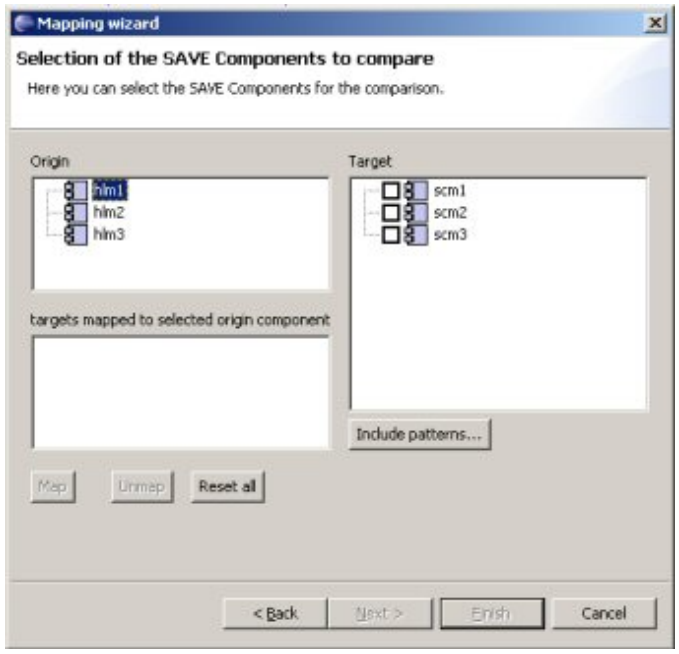

Figure 6-4 Case Study: SAVE Evaluation Mapping

Every component of the SCM is mapped to the corresponding one in the HLM, i.e. scm1 to hlm1, scm2 to hlm2 and scm3 to hlm3. The following figure shows the result of the evaluation.

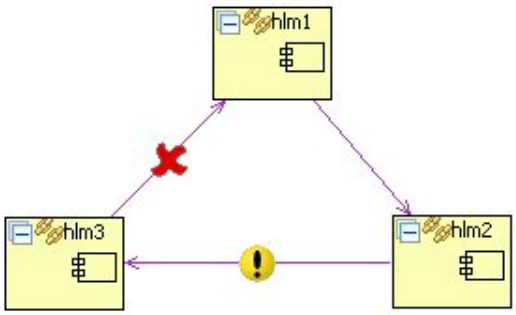

Figure 6-5 Case Study: Evaluation Result

The model generated during the evaluation process is always based on the high level model, which means that the components are the ones of the high level model. The relation between hlm1 and hlm2 exists and carries no additional mark which marks the type convergence. The relation between hlm2 and hlm3 does not exist in the HLM but was adopted from the SCM and carries an exclamation mark which indicates the type divergence. The last relation carries a red cross which indicates absent relations, i.e. it was planned in the HLM but not realized in the SCM.

7 Conclusion

This paper presents a solution to evaluate existing software systems against a planned architecture. It was developed on the basis of the r*eflexion models* as a plug-in for the *Eclipse Platform*. Therefore the *SAVE*-plug-in addresses the problem of potentially occurring discrepancies between a planned architecture and the realization.

The *SAVE Core Model* contains all data structures necessary to build a model which represents the architecture of a software system. Logically it can be split into three different entities, namely the *SAVEComponent Model* which represents the system on a high level of abstraction, the *FSModel* which represents a system very close to the implementation and the *SAVEFSConnector* which builds a mapping between the two mentioned models.

The evaluation is used to identify discrepancies as described above. The process can be subdivided into three phases. During the abstraction relevant facts are extracted from an existing software system and a model on high level of abstraction is generated. Two models on the same level of abstraction are a precondition to perform an evaluation. The generated model is the *Source Code Model (SCM),* the planned architecture the *High Level Model (HLM)*. Further more a mapping between the components of the SCM and the HLM has to be built. This is done in form of the *SAVEEvaluationMapping*. During the evaluation process one of the following *SAVEEvaluationTypes* is assigned to every relation: *convergence* if the relation exists in the SCM as well as in the HLM, *divergence* if the relation exists in the SCM but was not planned in the HLM or *absence* if the relation was planned in the HLM but not realized in the SCM.

The structure of the plug-in can be split into three different subsystems. First this is *de.fhg.iese.pulse.common* which contains data structures that can be used cross project according to the product line approach. d*e.fhg.iese.pulse.fe* contains data structures used for the fact extraction from projects of different programming languages and *de.fhg.iese.pulse.SAVE* such that realize the core features of the SAVE-plug-in.

Moreover the characteristics of the fact extraction with the help of the *Java Development Tools* provided by the Eclipse SDK were explained, with which compilation units can be parsed and a syntax tree is generated, that can be traversed and used to extract relevant facts.

In addition a case study was presented which demonstrates the evaluation of rudimentary software architecture.

References

[AL04] Arthorne, John ; Laffra, Chris: Official Eclipse 3.0 FAQ. Addison-Wesley Professional, 2004. ISBN 0321268385

[BCK03] Bass, Len ; Clements, Paul ; Kazman, Rick: Software Architecture in Practice 2nd Edition. Addison-Wesley Professional, 2003. ISBN 0321154959

[BSM+03] Budinsky, Frank ; Steinberg, David ; Merks, Ed ; Ellersick, Raymond ; Grose, Timothy J.: Eclipse Modeling Framework. Addison-Wesley Professional, 2003. ISBN 0131425420

[Ecl02] eclipse project FAQ. http://www.eclipse.org/eclipse/faq/eclipse-faq.html. nov 2002

[Ecl03] Eclipse Platform Technical Overview. http://www.eclipse.org/whitepapers/eclipse-overview.pdf . feb 2003

[EMF06] Eclipse Modeling Framework. http://www.eclipse.org/emf/. mar 2006

[GB03] Gamma, Erich ; Beck, Kent: Contributing to Eclipse: Principles, Patterns, and Plugins. Addison Wesley Longman Publishing Co., Inc., 2003. ISBN 0321205758

[KLMN06] Knodel, Jens ; Lindvall, Mikael ; Muthig, Dirk ; Naab, Matthias: Static Evaluation of Software Architectures. 10th European Conference on Software Maintenance and Reengineering (CSMR 2006), Bari, Italy.

[KS03] Koschke, Rainer ; Simon, Daniel: Hierarchical Reflexion Models. In: WCRE, 2003, S. 36 – 45

[MFK04+] Miodonski, Paul; Forster, Thomas ; Knodel, Jens ; Lindvall, Mikael; Muthig, Dirk : Evaluation of Software Architectures with Eclipse / Fraunhofer Institute for Experimental Software Engineering. 2004. Technical Report

[MNS01] Murphy, G.C. ; Notkin, D. ; Sullivan, K.J.: Software Reflexion Models: Bridging the Gap between Design and Implementation. In: IEEE Transactions on Software Engineering 27 (2001), Nr. 4, S. 364 –

380

[NFK+05] Naab, Matthias ; Forster, Thomas ; Knodel, Jens ; Muthig, Dirk : Evaluation of Graphical Elements and their Adequacy for the Visualization of Software Architectures / Fraunhofer Institute for Experimental Software Engineering. 2005. Technical Report

[Sof06] How Do You Define Software Architecture? http://www.sei.cmu.edu/architecture/definitions.html . mar 2006

Document Information

Title:	The SAVE Plug-in - Internal Data Model and Architecture Evaluation Functionality
Date:	June 8, 2006
Report:	IESE-063.06/E
Status:	Final
Distribution:	Public