University of Kiel

Department of Computer Science

Software Engineering Group

Reconstructing Software Architectures using the Code- and Structure Package of the Knowledge Discovery Meta-Model

Bachelorthesis

2010-02-23

Written by: Florian Fittkau
born on 1987-01-14 in Kiel

Supervised by: M. Sc. Sören Frey
Prof. Dr. Wilhelm Hasselbring

Abstract

Software maintenance consumes 60 % to 80 % of the total life cycle costs of a software system [4]. The maintenance process involves the understanding of the underlying system by 50 % to 90 % [43]. Thus, the understanding of a system is essential to reduce the total life cycle costs. Software Architecture Reconstruction (SAR) can yield the needed comprehension of an actual software system, e.g. if the knowledge about the internal structure was lost over time. However, reconstructing the architecture of an existing system is not trivial. Many research on this field has been done. Recently the Object Management Group (OMG) proposed a new meta-model, namely the Knowledge Discovery Meta-Model (KDM), for representing the information contained in a software system. In this thesis I present a newly developed SAR tool, KDM Architecture Discoverer (KADis), which utilizes this meta-model. Furthermore, I present an made evaluation of different data formats including KDM for SAR. In addition, I propose a new categorization of SAR activities that focuses on the basic understanding of SAR.

Contents

1 Introduction **1**

2 Approaches to Software Architecture Reconstruction **3**
2.1 Common Phases in Software Architecture Reconstruction Processes 3
2.2 Phase Data Extraction 5
2.2.1 Static Analysis 6
2.2.2 Dynamic Analysis 6
2.3 Phase Repository Storing 7
2.3.1 KDM 7
2.3.2 GXL 7
2.3.3 RSF 9
2.3.4 FAMIX 10
2.4 Phase Abstraction 11
2.4.1 Decomposition 11
2.4.2 Class Hierarchies 12
2.4.3 Class Diagrams 12
2.4.4 Interfaces 12
2.4.5 Design Patterns 12
2.4.6 Conformance 13
2.4.7 Feature Location 13
2.4.8 Use Case 13
2.4.9 Configuration 13
2.4.10 Object Traces 14
2.4.11 Component Interaction 14
2.4.12 Process Interaction 14
2.4.13 Object Interaction 14
2.4.14 Conceptual 14
2.4.15 Responsibility 15
2.4.16 Build Process 15
2.4.17 Files 15
2.4.18 View Integration/Combination 15
2.5 Further Approaches 15
2.5.1 Symphony 16
2.5.2 Focus 17
2.5.3 Quality Attribute Driven SAR 19

3 Knowledge Discovery Meta-Model **21**
3.1 Structure of KDM 21
3.2 Example for a KDM-conform File 23

4 Evaluation of different Interchange Formats for Software Architecture Reconstruction 26
4.1 Assessment criteria . . . 26
4.2 KDM . . . 27
4.3 GXL . . . 28
4.4 RSF . . . 28
4.5 FAMIX . . . 29

5 Development of KADis 30
5.1 Features . . . 30
5.2 Design . . . 30
5.3 Activities in KADis' SAR process . . . 30

6 Evaluation of KADis 32
6.1 Completeness Evaluation with JPetStore . . . 32
6.1.1 Assessment Criteria . . . 32
6.1.2 Discussion . . . 33
6.2 Performance Evaluation . . . 34

7 Related Tools in Software Architecture Reconstruction 36
7.1 ARMIN . . . 36
7.2 Rigi . . . 36
7.3 Moose . . . 36
7.4 MoDisco . . . 36

8 Conclusion and Future Work 38
8.1 Conclusion . . . 38
8.2 Future Work . . . 38

References 40

A Acronyms i

B Glossary ii

C JPetStore elements iii

D Functional Specification v

E Design Specification xxxii

F Attachments lxxx

1 Introduction

Many companies use old software systems. Over the years new requirements to the software arise. These requirements are often implemented without explicit knowledge of possible side-effects. The knowledge about the exact architecture was lost over time. Important reasons for this fact are a missing or outdated documentation and architectural erosion. This often results in a higher error rate of the software.

In many cases a new development of the software is refused or not economic. Software Architecture Reconstruction (SAR) provides the needed understanding for maintaining the existing software system. This enables an improvement or migration of the existing software architecture, for instance. However, SAR is not trivial. Often many million lines of source code have to be analyzed, abstracted, and then presented in an appropriate way. Many research has been done on this field and many terms were introduced. Some SAR approaches use the term *redocumentation* of software architectures to emphasize the creation or update of a documentation. Other approaches use the term *recovering* software architectures to describe the fact that in many cases the software architectures are outdated and have to be recovered. In this thesis, I will use the term *reconstruction* of software architectures to focus on the process of reconstructing the views of the software architecture.

On the one hand, SAR needs a storage format for often million lines of source code and the data which is abstracted from it. On the other hand, SAR includes many different phases and thus needs different tools that cover different phases. For reasons of interoperability these tools need a common data interchange format that has to represent all saved information. A data format that enables both the data interchange and the data storage makes the development easier. The Object Management Group has defined a common data interchange format named Knowledge Discovery Meta-Model (KDM). The thesis will evaluate this data format in the context of SAR.

In this thesis, I show a new categorization of SAR activities by grouping them in identified common phases. Furthermore, I conducted an evaluation of KDM in comparison with other common data interchange formats in the context of SAR. Additionally, I developed KDM Architecture Discoverer (KADis), which uses KDM for the generated output, to evaluate the applicability of KDM. KADis limits to the code and structure package of KDM which is a restriction to the static facts of the software system that should be analyzed.

Related work lists other categorizations of SAR approaches. For example, one categorization can be performed on basis of the level of automatism. Another tool in SAR, which uses a meta-model for its data storage, is Moose. The used format of Moose, FAMOOS Information Exchange Model (FAMIX), is part of the format evaluation.

The remainder of the thesis is structured as follows. Section 2 outlines the identified common phases in SAR approaches. Afterwards, Section 3 describes KDM in detail. Then, Section 4 presents an evaluation of different data interchange formats in

SAR. A developed prototype, named KADis, for using the KDM is discussed in Section 5. The following Section 6 evaluates KADis. Section 7 describes different related SAR tools. The final Section 8 concludes important points of the thesis and defines the future work for advancing KADis.

2 Approaches to Software Architecture Reconstruction

A SAR process can be performed manually, semi automatically, or fully automatically. In the manual processing the reverse engineer analyses the source code and has only limited tool support. The tools can be syntax highlighters or visualization tools, for instance. In the semi automatic process the SAR tool proposes a possible solution and the reverse engineer has to judge if the suggestion of the tool is correct. If the suggestion is not correct the SAR tool tries to find a different solution. The fully automatic process does not need human interaction. The human interaction is done implicitly by internal assumptions about the correct solution.

This Section presents different techniques in the context of SAR by categorizing them into common phases. The Section is not thought as a broad survey of every approach available in SAR. For this purpose Ducasse et al. have written a survey [10], for instance. This Section intends to give a basic understanding of SAR.

2.1 Common Phases in Software Architecture Reconstruction Processes

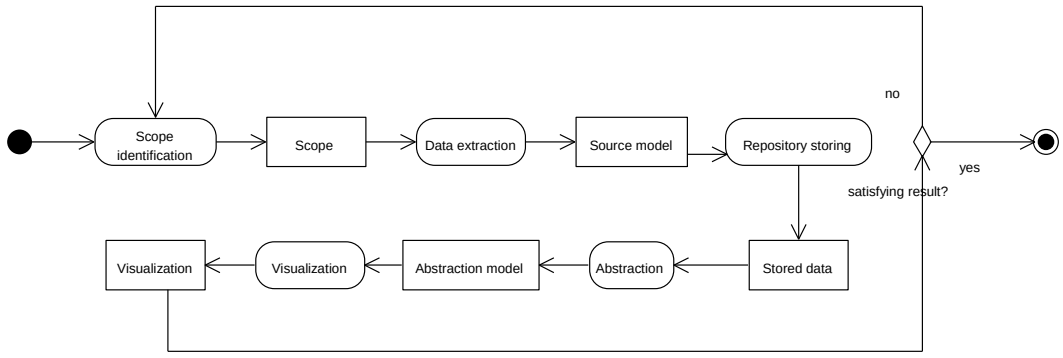

Figure 1: Overview of the identified common phases and sequence in the SAR process

I identified the common phases and sequence shown in Figure 1 and extracted them from different approaches and methods used in SAR [5, 10, 21, 27, 30, 38, 39, 45]. The created overview, presented in the following, focuses on the SAR process itself and thus enables a better understanding of the different phases involved in SAR. Other categorization can be made, too. For instance, the categorization formerly mentioned into manual, semi automatic, and fully automatic approaches.

The overview starts with the scope identification. Scope identification recurs in many SAR approaches. If the scope identification is done only implicitly, the whole system is selected. However, in most cases a special subsystem is the main interest to focus on.

Data extraction, abstraction, and presentation are well known steps in SAR [39]. The data needed for SAR, like classes or packages, has to be fetched from software artifacts. Then, the data has to be abstracted to enable different views onto the system and then has to be presented. The presentation is almost always done as visualization. Therefore, the last step is not called presentation but visualization. Between data gathering, abstraction, and visualization the data has to be saved to enable more flexibility in presentation. However, this step is not trivial. By saving information about the reconstructed software system different purposes shall be achieved. These are, for instance, information exchange with other tools supporting reusability or enabling the definition of an easy query language for changing the focus of the visualization.

The remainder of the Section describes the common phases. A more detailed view on *data extraction*, *repository storing*, and *abstraction* is presented in Section 2.2 to 2.4. *Scope identification* and *visualization* are not considered in detail because this would go beyond the scope of the thesis. Finally, Section 2.5 presents SAR approaches that define an extra step to emphasize different aspects like, for instance, quality attributes.

Scope identification

The user can select the software artifacts to focus on in the SAR process. This selection of important facts, like different subsystems or quality attributes, is called scope identification. By changing the scope on which software artifacts to focus on, the process can comprise many iterations. Each iteration reveals new knowledge about the software. It is easier to understand a huge system by considering higher abstraction levels first. To minimize the amount of data, e.g. classes that can be clustered into components, on the higher abstraction levels, identifying the scope can act as a filter for subsystems.

Data extraction

The data extraction phase collects the data, like classes, functions or build files, used for SAR, e.g. by external parsers. One example of an external parser is the Source Navigator NG [36], which is used for data extraction in KADis (Section 5). The output of this phase is called source view or source model.

Usually, not all information about a software architecture is contained explicitly in the source code. Knowledge can be extracted from other software artifacts, too. These are, for example, build files, tests, and configuration files. Other methods can even extract information from documentation. However, not all knowledge about the software architecture is contained in files. Interviews with software architects and other stakeholders can be done for revealing the business process, for instance.

For more details and further methods for data extraction refer to Section 2.2.

Repository storing

The data has to be saved somewhere. Typically this is done in a repository. This repository can be a file or a database, for instance. Many SAR tools use databases. Databases make the querying of information faster and easier, and a query language for selecting items is

built-in. Meta-models like KDM can be used as a data interchange format and a data format for data storage as well. They can be used natively by saving files or can be translated into a database schema for usage in a database. Possible data formats are described in Section 2.3. After the phase has finished, the data of the *data extraction* activity is stored in a repository.

Abstraction

The data extraction can produce a big amount of data. Classes and functions represent a huge part of that data. However, in a big software system considering classes or functions is not useful as a starting point. A more coarse view has to be presented or in other words the level of abstraction has to be raised. Therefore, different abstraction methods have been proposed. Clustering and forming modules by special names, for instance. Section 2.4 describes different methods for abstraction. After performing the abstraction phase an abstraction model is available.

Visualization

The data has to be visualized for easier reading and understanding. Many tools use their own domain-specific language (DSL). Some of the tools for SAR use common graphical DSLs, Unified Modeling Language (UML) in most cases, for attaining a common understanding of the displayed data.

Different layouts and display styles in visualization can be used to emphasize different aspects of the software architecture. For instance, a hierarchical view highlights the hierarchical aspect of subsystems. The output of this phase is a visual representation of the views.

2.2 Phase Data Extraction

For extracting data there exist two distinct classes of approaches, namely the static and dynamic approaches [27, 28]. Static approaches basically provide an overview of static information extracted from source code like classes or packages, for instance. Dynamic approaches enable information extraction at runtime, for instance the constructor calls. Both classes of approaches have their advantages and disadvantages. Static and dynamic approaches can be too fine granulated to give a good comprehension of the system and the dynamic approaches additionally do not always yield all data. Therefore, a combination of static and dynamic approaches is used in practice. This gives the need of having a merge between them. The merge of two views is called View Fusion. The remainder of Section 2.2 presents different static and dynamic approaches.

2.2.1 Static Analysis

There are different static analysis methods which differ in precision, scalability, and effort. In practice mostly a mixture of them is used, to enable the best ratio between the mentioned attributes. In the following the different static approaches are presented.

Manual analysis

The reverse engineer analyses the source code manually with only restricted tool support. He looks at the source code by hand. Many data can be fetched from the system by looking at the directory structure, for instance. However, in big projects the manual analysis proves only partially useful because it is accompanied by a big effort.

Lexical analysis

Lexical analysis converts a given sequence of characters to tokens. There are many tools available that perform lexical analysis. *Grep* searches for regular expressions, for example. A regular expression or a so called lexical pattern is, for example, the keyword "include" in C++.

Syntactic analysis

Syntactic analysis or in other words parsing defines the process of analyzing a given text, made of tokens, to determine its underlying grammatical structure on the basis of a specific formal grammar. The tools that perform this task are called parsers. They typically construct a syntax tree. Then the user can query the syntax tree for certain patterns to focus on different aspects of the result.

Semantical analysis

Semantical analysis can be done by parsing the context or variable names. After the parsing the control and data flow analysis can help to improve the results.

2.2.2 Dynamic Analysis

In dynamic analysis different methods and tools can be used to get dynamic runtime information from the system. For example, there exist profiling tools (e.g. *gprof*) or code instrumentation tools. Dynamic information can be very useful since static analysis can not recover late bindings easily. Examples for late binding are polymorphism, function pointers, and runtime parametrization. Dynamic information becomes essential in a multiprocess system that creates threads dynamically at runtime. Many tools can trace the execution path of the program and analyze it. For instance, Kieker [33] is a tool for continuous monitoring, analysis, and visualization of Java software behavior.

2.3 Phase Repository Storing

As mentioned earlier the input and output data formats should be used for the format of the repository too and thus this Section describes data formats. At the beginning of research in SAR nearly every developed SAR tool had its own data format. In the past years different efforts to a common data interchange format have been made. There are two major classes of data interchange formats, namely meta-models and graph based data formats. KDM and FAMIX belong to the meta-models. Graph eXchange Language (GXL) and Rigi Standard Format (RSF) are graph based data interchange formats. The Sections 2.3.1 to 2.3.4 describe the different data formats that were designed to become a common data interchange format. In Section 4 an evaluation of these different data formats is presented.

2.3.1 Knowledge Discovery Meta-Model

KDM was developed by the Object Management Group [17]. This consortium has already specified the well known UML. KDM was designed with the goal of creating a common data interchange format for SAR tools, where every element is clearly defined. Section 3 presents it in detail.

2.3.2 Graph eXchange Language

GXL is a standard data format for exchanging graph representations. The motivation for creating GXL was to enable interoperability between different tools like extractors, abstractors, and visualizers. With a common data interchange format a powerful reverse engineering workbench can be built, for instance. At the Dagstuhl Seminar "Interoperability of Reverse Engineering Tools" in January 2001, GXL was ratified as a standard exchange format in reverse engineering [49].

Tuple Attribute Language (TA) [23], GRAph eXchange format (GraX) [12], and the graph format of PROGRES [35] were merged to create GXL. Furthermore, the authors added concepts for handling hypergraphs and hierarchical graphs. In addition, GXL includes different ideas from Relation Partition Algebra (RPA) [13] and RSF [19]. GXL was influenced by several other formats used in graph drawing, e.g. daVinci now known as uDraw(Graph) [16] and GML [22]. Thus, GXL can be seen as a generalization of these formats.

GXL offers the possibility to exchange different kinds of graphs. These kinds may be typed, attributed, directed, ordered graphs including hypergraphs, and hierarchical graphs. Furthermore, the graph schemas can be exchanged as metaschemas. For example, UML diagrams can be represented and exchanged by supplying the appropriate UML metaschema, which defines the semantic description of UML. Therefore, GXL can handle different types of graphs and its underlying semantic.

Figure 2 shows an example graph taken from Holt et al. [24]. This graph is an attributed, typed, and directed graph with two types of edges and two types of nodes. The

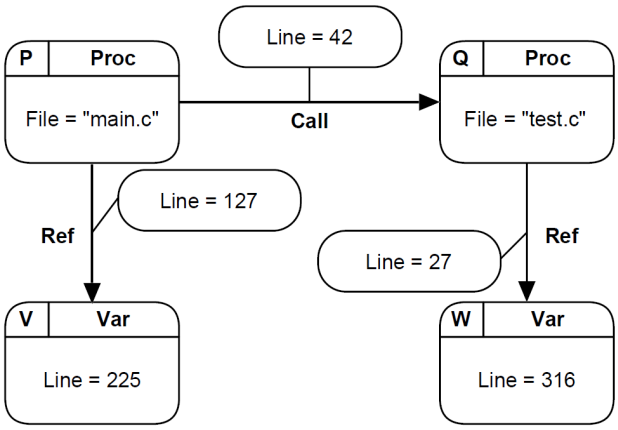

Figure 2: GXL example graph

example can be interpreted as follows. On line 42 procedure P calls procedure Q. The procedure P is stored in file main.c and procedure Q is stored in file test.c. P references variable V in line 127 and Q references variable W in line 27. The variable V is defined in line 255 and the variable W is defined in line 316. The edge (P,Q) has type *call* and a *line* attribute with value 42.

```
1  <gxl>
2    <node id="P" type="Proc">
3      <attr name="File" value="main.c"/>
4    </node>
5    <node id="Q" type="Proc">
6      <attr name "File" value="test.c"/>
7    </node>
8    <node id="V" type="Var">
9      <attr name "Line" value="225"/>
10   </node>
11   <node id="W" type="Var">
12     <attr name "Line" value="316"/>
13   </node>
14   <edge begin="P" end="Q" type="Call">
15     <attr name "Line" value="42"/>
16   </edge>
17   <edge begin="P" end="V" type="Ref">
18     <attr name "Line" value="127"/>
19   </edge>
20   <edge begin="Q" end="W" type="Ref">
21     <attr name "Line" value="27"/>
22   </edge>
23 </gxl>
```

Listing 1: "GXL example XML"

Listing 1, again taken from Holt et al., displays the corresponding XML file of the GXL example graph seen in Figure 2. Basically, there are 2 different types of elements. The first one is the *node* type, which represents nodes in the graph. The second one is the

edge type, which stands for the edges in the graph. These types can contain *attr* subtags, which indicate that the type has attributes.

This simple example does not use the full Document Type Definition (DTD) of GXL. For more information about the possible tags and GXL refer to Winter et al. [49].

2.3.3 Rigi Standard Format

Rigi [51] is a workbench tool for SAR. The *rigiedit* component of the workbench specifics a data import format. This format is called RSF [50]. RSF has two major dialects. These are unstructured and structured RSF. Usually, external tools use the unstructured RSF and *rigiedit* saves the provided graph as structured RSF.

RSF bases on tuples. The mainly used unstructured RSF is 3-tuple based. The structure of the 3-tuple is: verb subject object. Three different cases are allowed.

The first case is representing an arc between two nodes. The structure looks like *arcType startNodeName endNodeName*. Listing 2 presents an example for this structure. The first line represents the fact that the *main* function calls the *createArray* function. The last line shows that the *createArray* function creates or accesses the *data ARRAY*.

```
1 call main createArray
2 data createArray ARRAY
```

Listing 2: "RSF arc example"

Attributes binding is the second case for the allowed structure of 3-tuple based unstructured RSF. The structure is: *nodeAttribute nodeName attributeValue*. Listing 3 shows an example for attribute bindings in RSF. The lines declare *createArray* occurred in a *file* named *array.c* at line 10 in the file.

```
1 file createArray ``array.c''
2 lineno createArray 10
```

Listing 3: "RSF attribute binding example"

Node type binding is the last case for the allowed structure. The structure is defined as: type *nodeName nodeType*. Listing 4 presents an example for types of node bindings in RSF. The node *createArray* is binded to the type *Function* and the node *ARRAY* is binded to the type *Data*.

```
1 type createArray Function
2 type ARRAY Data
```

Listing 4: "RSF node type binding example"

With these 3 different types of 3-tuple based RSF a graph can be represented. Thus, enabling the exchange of graph information. The structured RSF format is not described here because it is basically only used internally by *rigiedit*.

2.3.4 FAMIX Meta-Model

The workbench tool Moose utilizes the FAMIX meta-model [9, 42]. FAMIX was developed with the goal of describing object-oriented software systems at the program entity level. The model has been designed for facilitating language independence, extensibility, and information exchange.

The inventors developed FAMIX as an alternative to UML in version 1.2 for round-trip engineering [8]. They claim that UML in version 1.2 has different shortcomings in giving seamless integration between design diagrams and source code, between modeling and implementation. Therefore, UML is lacking the ability of some important concepts to generate source code from models and vice-versa. These concepts are namely "method invocation" and "attribute access". UML could have been extended at that time, but this would make data exchange between different tools very complicated and not standardized. The authors wanted to guarantee tool interoperability. Hence, they developed FAMIX.

FAMIX bases on the CDIF [15] transfer format. CDIF is an industrial standard for transferring models and provides standard plain text encoding. Therefore, enabling human readability. XMI [18] was also considered for exchanging information but, when development started, XMI was too premature.

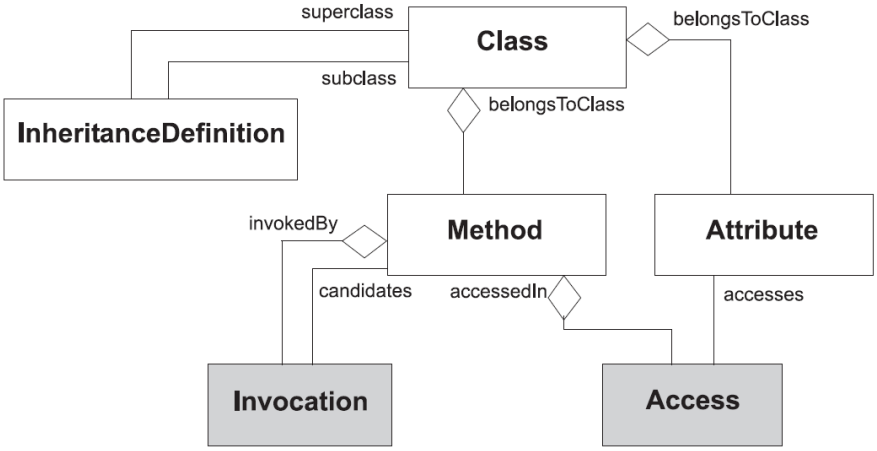

Figure 3: FAMIX core model

Figure 3 taken from Demeyer et al. [9] represents the core model of FAMIX. The core model consists of the main entities utilized in the object orientation paradigm. These are class, method, attribute, and inheritance definition. Reengineering needs the invocation and access entities. Invocation means that one method calls another method and access represents the fact that a method accesses an attribute. These two extra entities are needed by reengineering tools for dependency analysis, metrics computations, and other

reengineering operations. For example, the invocation entity can be used for evaluating which method is never invoked.

2.4 Phase Abstraction

Data extraction fetches a great amount of information, e.g. classes and functions. In a big software system classes must be clustered to modules to get a better understanding of the system, for instance. Software architecture building differentiates different views. These views must be reconstructed. There are many definitions of the typical viewpoints onto a system. The following Section uses the viewpoints categorization by Koschke [28] that bases on the definition of Clements et al. [6]. This categorization was chosen because some of these viewpoints, namely the configuration, conceptual, build, and files viewpoint, can be directly mapped to the corresponding KDM package. The other viewpoints can be mapped to parts of the code, action, and structure package of KDM. Thus, the selection gives a first hint to generate KDM packages. The remainder of the Section describes these different viewpoints and sketches some possible methods for reconstructing the viewpoints. The Section is inspired by the survey of Koschke [28] that defines the used categorization of viewpoints. In his survey many examples and references can be found. However, this Section focuses on providing an overview for reconstructing the viewpoint and the purpose of the reconstructed viewpoint in SAR. Thus, giving a hint when to reconstruct each viewpoint.

2.4.1 Decomposition

The most research has been done in the derivation of the structural decomposition. An example for this inference is the grouping at lower abstraction levels like classes, variables, functions into modules. At higher levels, modules can be clustered to subsystems and layers. For grouping, static dependencies, method calls, and variable accesses are used, for example. Dynamic dependencies can be used too.

There are different approaches for grouping. Software clustering is the most popular one. Borrowed from biology the clustering of animals by similarity, the classes can be clustered by different attributes. Different definitions of similarity lead to different grouping. Other methods see grouping as a partitioning problem which can be solved by minimizing coupling and maximizing cohesion between entities. Grouping bases on semantic. Therefore, the process can only give suggestions for the best grouping based on different assumptions. Thus, these methods are mostly incremental techniques. The user has to interact with the process to find the best match for grouping. The purpose of this viewpoint is to find components that can be clustered to less complex components. This is done until different subsystems are visible. Thus, enabling an overview of the reconstructed system.

2.4.2 Class Hierarchies

At first view, class hierarchies can be trivially retrieved from object-oriented programming languages. However, these class hierarchies may not be optimal. The optimal hierarchies would only contain used methods and variables. By using dynamic approaches the unused methods can be identified. There exist also approaches that build complex static method call trees that can find the unused methods too. Class inheritances reveal the internal structure of the classes. The concrete class can inherit from an abstract class which by itself can inherit from another abstract class. Therefore, class hierarchies can be used to understand the abstract base classes that many classes can inherit from. Thus, the class hierarchies viewpoint can reduce the complexity and give an overview of the structure again.

2.4.3 Class Diagrams

Class diagrams constitute another viewpoint. They have different association types like aggregation or composition. Class diagrams display the dependencies between classes beyond inheritance relation. There are different methods for reconstructing the class diagrams from source code. For instance, a class diagram can be generated by parsing the import statements of a class. Though, this example would need an object-oriented programming language like Java. Nevertheless, class diagrams can even be extracted from procedural languages by grouping functions to classes. For example, class diagrams are useful when a layer of a software system has to be understood.

2.4.4 Interfaces

There exist different dependencies between modules. A module provides and requires access to other modules. This is supported by the interface concept in Java, for instance. Even if the interfaces are not explicit in a programming language, they can be fetched from coarse import directives. Maybe not every import is used or not all exports are used. Here the problem of finding the optimal interface structure arises again.

Interfaces may define preconditions and postconditions and valid sequences of interaction for correct collaboration of modules. This concept is also known as programming by contract. A very important principle in a big system with different modules is to guarantee proper invocation between subsystems. Thus, the interfaces viewpoint most often shows the communication between modules.

2.4.5 Design Patterns

Design patterns are an essential concept for software engineering. First research enabled only the detection of structural patterns. Later research included behavioral and creational patterns too. Static approaches use pattern matching. The design pattern is matched in a graph representation of the class model or abstract syntax tree. Automatic validation

can then be done by utilizing dynamic execution traces. Pattern matching underlies the NP hard problem of finding isomorphic subgraphs [1]. Therefore, approximative methods have been proposed. In approximative methods false positive and the true negative match exist. These false matches can be caught by interaction with the user. By detecting the design patterns the reverse engineer gets a quick understanding of the module in which the design pattern is used.

2.4.6 Conformance

Interesting for, e.g., maintenance purposes is the conformance between the implemented architecture and the intended architecture. The intended architecture is often called the hypothetical view. A popular approach for this is the reflexion model by Murphy et al. [31]. They define the following states between the two models. A *convergence* is a match between the hypothesized and concrete model. A *divergence* is an element which is contained in the concrete model but not in the hypothesized model. An *absence* is an element which is included in the *hypothesized model* but not in the concrete model. With the conformance viewpoint the reverse engineer can detect architectural erosion and violation of the used architectural style.

2.4.7 Feature Location

Many modules contribute to a product feature. Often it is unknown which modules realizes which feature. To locate the implemented-by relation a global static dependency graph can be built and the reverse engineer manually searches for the feature in the graph. Dynamic analysis supports the reverse engineer with an execution trace when the feature was executed. This trace provides hints to which module or class is associated to the feature. With the feature viewpoint the reverse engineer can extract the modules that take part in a feature and then migrate them to a new architecture, for instance.

2.4.8 Use Case

A Use Case can comprise many features. Therefore, the detection of Use Cases relates to feature location. A static approach proposed by Lucca et al. [29] starts with an input statement till it finds an output statement. The search creates a method-message graph. A Use Case equals one path in it. Like in the feature location, the reverse engineer can extract the modules that take part in a Use Case and then migrate them to a new architecture, for example.

2.4.9 Configuration

Source code is often configured with preprocessor directives. This can lead to different components at higher abstraction levels. The configuration viewpoint shows which line of code is compiled with which directive. Especially, this viewpoint presents the elements

that change on higher abstraction levels. It can be used to identify the components that are included in a configuration, for instance.

2.4.10 Object Traces

A trace is a record of the execution of software that shows the sequence of operations executed. This viewpoint can be reconstructed statically and dynamically. Dynamic analysis simply records the execution of the program while running. Static approaches try to find every possible execution path. The object trace viewpoint should be reconstructed, e.g., if an object is created and passed multiple times to understand the processing of the object.

2.4.11 Component Interaction

Components are modules that encapsulate a set of related concerns and they can interact with other components. A component can be, for example, a database. The component interaction viewpoint shows the concrete interaction with other components. Hence, this viewpoint shows among other things the dependencies to other components. Prior to the reconstruction of this viewpoint the components in the system have to be identified. The interaction with other components can be analyzed by recording the interactions as a trace in dynamic analysis. Other approaches can detect the interaction with static analysis by following the call chain. The interaction between a component and another component can be interesting, for example, for the purpose of replacing a component.

2.4.12 Process Interaction

Process interaction is coupled with the component interaction. However, process interaction is harder to analyze statically because processes can be distributed. Nevertheless, the interaction between processes can be easier reconstructed dynamically by recording the communication. This viewpoint can be useful, e.g., when two different processes, which were located on the same server, shall be run on two different machines.

2.4.13 Object Interaction

The object interaction viewpoint describes the messages that are sent between objects. This kind of interaction can again be analyzed by static and dynamic analysis. The dynamic analysis records a dynamic trace of the object interaction but this trace does not need to be complete. This viewpoint can assist a reverse engineer to understand, e.g., the dependencies of objects.

2.4.14 Conceptual

The conceptual viewpoint shows how the software functionality is mapped to components and connectors. The implemented-by relation can be seen with this viewpoint. The con-

ceptual viewpoint is used to understand how the software system is achieving conformance with the specification.

2.4.15 Responsibility

Every source code file has at least one programmer as author and therefore this programmer is responsible for the file. The responsibility viewpoint contains a relation that maps from a source code file to a developer. This relation can be reconstructed, e.g., by analyzing the copyright notices like done by Bowman and Holt [3]. With the responsible relation the expert for the file can be identified.

2.4.16 Build Process

The build process of a huge software system can become very complicated. The build process viewpoint represents the configuration, data, activities, and strategy of the build system. The build view was proposed originally by Tu and Godfrey [44]. With this viewpoint the reverse engineer can understand the build process which becomes necessary when a component has been refactored and the build process must be adapted, for instance.

2.4.17 Files

Every line of source code is saved in a file. The file viewpoint can be obtained by static analysis. It represents the physical structure of source code and can be useful, when the physical structure of the given source code should be refactored.

2.4.18 View Integration/Combination

The combination of two views can be prolific. For example, the static view can be combined with a dynamic view resulting in more available information that perhaps one of these views could have leaked. Thus, view integration can enable a better coverage of information in a view.

2.5 Further Approaches

There are differing approaches to the common phases described in Section 2. Many of these approaches add a new phase or aspect to the common phases. Symphony, Focus, and Quality Attribute Driven SAR are examples for these differing approaches. Symphony, described in Section 2.5.1, focuses on the prior problem elicitation activity in a SAR process. Focus emphasizes the importance of incorporating architectural styles in SAR and is illustrated in Section 2.5.2. Quality Attribute Driven SAR utilizes SAR for the determination of quality requirements. Section 2.5.3 describes this approach.

2.5.1 Symphony

Symphony [45] is a view-driven approach. The authors claim that most SAR approaches do not provide information about when to reconstruct a specific view. This gap should be filled by Symphony. Therefore, Symphony provides two different steps. The reconstruction design step is the first step and the reconstruction execution step is the second step. Both steps are used incremental. The reconstruction design step and execution step of Symphony is illustrated in the following Figure 4 taken from van Deursen et al. [45].

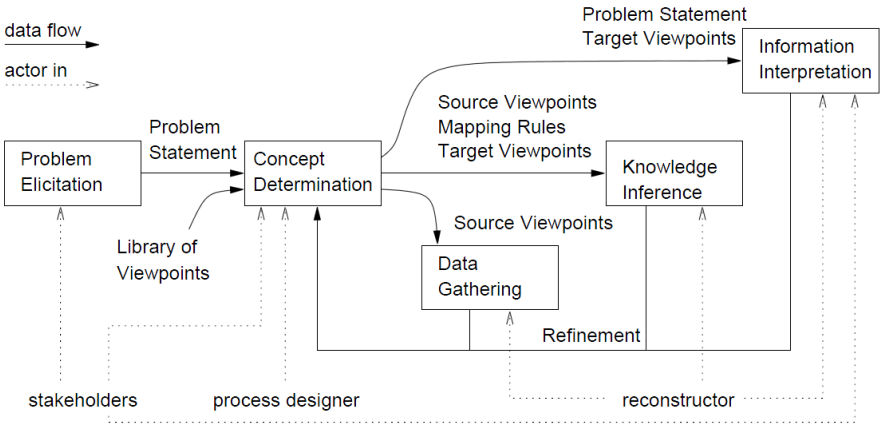

Figure 4: Symphony design step and execution step

Design step

The left part of Figure 4 shows the design step and the right part of Figure 4 illustrates the execution step. The outcome of the design step is a plan for reconstructing the software architecture. It consists of *problem elicitation* and *concept determination*. The *problem elicitation* activity collects all available information about the software architecture and elaborates the problem statement. For this information gathering, workshops or interviews with the available stakeholders that created the software are conducted in this activity. In addition, the relevant high-level documentation is summarized. Then, the *concept determination* activity identifies the viewpoints that need to be reconstructed and defines the mapping rules for the reconstruction that will be made in the execution step.

Execution step

Figure 5, again taken from van Deursen et al. [45], presents the reconstruction execution step of Symphony. This step consists of *data gathering*, *knowledge inference*, and *information interpretation*. These tree activities map basically to the *data extraction*, *abstraction*, and *visualization* phase described in 2.5. Though, in the *knowledge inference* activity the abstraction mechanisms base on the rules and viewpoints defined in the design step.

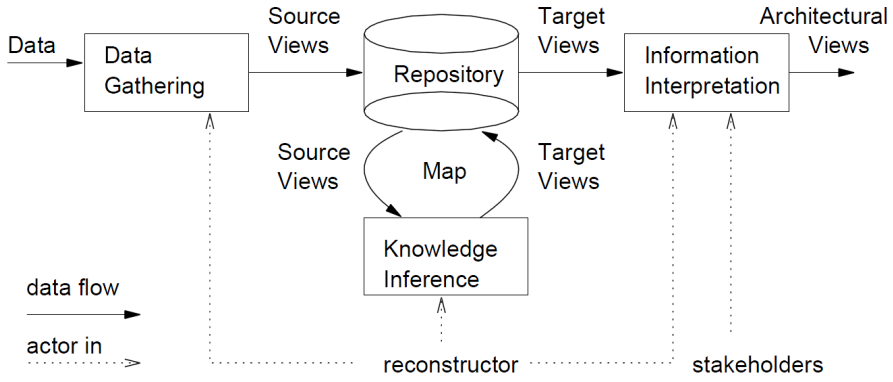

Figure 5: Symphony execution step

2.5.2 Focus

Focus is an approach from Medvidovic and Jakobac [30]. Medvidovic and Jakobac claim that most SAR approaches are heavy-weighted because they want to completely recover the software architecture. The authors of Focus wanted to provide a light-weight approach for software architecture recovery that includes the reconstruction of the architectural style. Their approach is semi-automatic and incremental. Focus has three unique facets. The first one is the fact that Focus uses a system's evolution requirements to isolate and incrementally recover only the components which are effected by the evolution. Therefore, enabling a focused view onto the system's parts that shall be changed. Secondly, Focus reconstructs not only the software components, but recovers the key architectural notions of software connector and architectural style. The last facet is the ability of Focus to refactor the system.

Focus conducts two different, interrelated steps. Theses are architectural recovery and system evolution. The steps are displayed in Figure 6 and 7 and are described in the remainder of the Section.

Architectural recovery step

The architectural recovery step is shown in Figure 6, which is taken from Medvidovic and Jakobac [30]. This step has the purpose to recover the actual architecture based on an idealized architecture in an incremental usage. The activities, described below, are separated into two categories: logical and physical architecture recovery. The logical architecture recovery starts with an idealized, high-level model of the software architecture, which is, for instance, inferred from the prior selected architectural style, and tries to refine the selected components by integrating more concrete details into the idealized architecture. The physical architecture recovery starts with the source code and tries to abstract it to get the actual components of the system. By incrementally applying the step the architecture becomes more and more consistent with the actual architecture.

The step seen in Figure 6 is composed of six activities, namely *Identify components*, *Propose idealized architectural model*, *Map components onto architecture*, *Identify key Use*

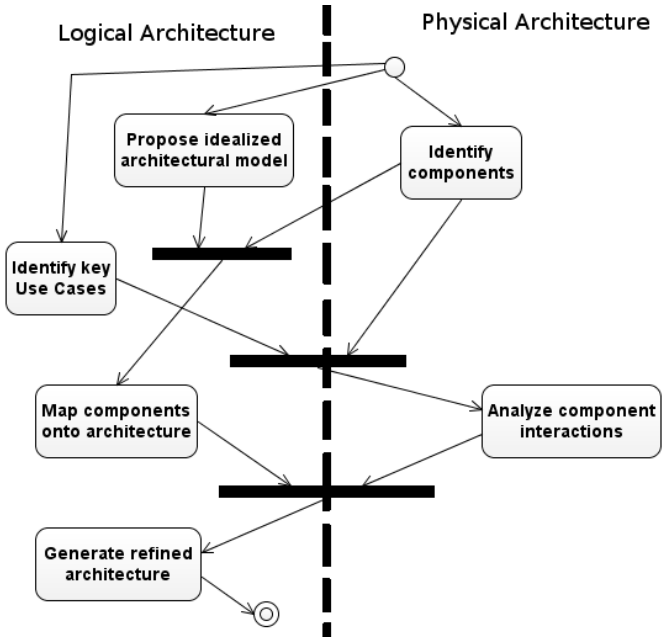

Figure 6: Focus architectural recovery step

Cases, *Analyze component interactions*, and *Generate refined architecture*. At first the *Identify components* activity gathers data need for the abstraction of components from the source code. Then the *Propose idealized architectural model* activity chooses an idealized architecture model. Different hints in the architecture, like GUI-based or Internet communication, can yield the architectural style and thus the needed idealized architecture model. Afterwards, the *Map components onto architecture* activity maps the identified components from the first step onto the idealized architecture. After this activity the key Use Cases are abstracted from the selected components in the *Identify key Use Cases* activity. The following *Analyze component interactions* activity analyzes the component interactions. Finally, the refined architecture is generated in *Generate refined architecture*.

System evolution step

The next step in Focus is the system evolution step shown in Figure 7, which is taken from Medvidovic and Jakobac [30]. In this step Focus modifies the application to satisfy the new requirement. The five activities, that are conducted in this step, are *Propose idealized arch evolution*, *Add / Modify components*, *Update component interactions*, *Generate evolved architecture*, and *Set the new focus*. These activities are all activities in a refactoring setting, which is out of the scope of the thesis. Therefore, the activities are only described briefly for completeness and understanding of the Focus approach.

The first activity *Propose idealized arch evolution* creates a high-level architecture evolution plan. The *Add / Modify components* activity then carries out the first step of the evolution plan in a semi-automatic way by interaction with the reverse engineer. Then, the next activity *Update component interactions* checks the component interactions and updates, if necessary, the interactions. After this activity the changes are integrated

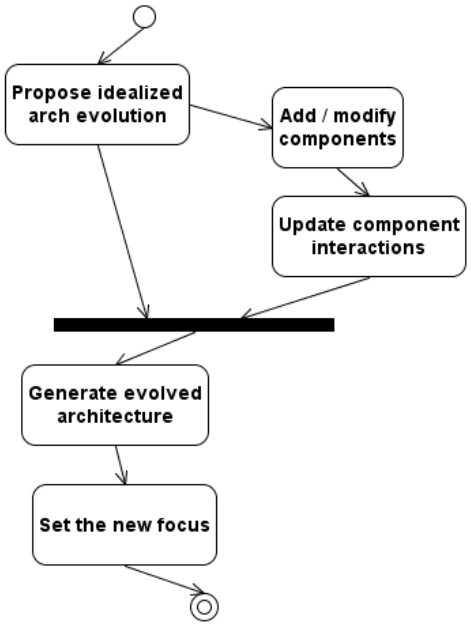

Figure 7: Focus system evolution step

into the original architecture. This is done in *Generate evolved architecture*. Finally, the *Set the new focus* activity decides whether or not the generated architecture consists of sufficient details to enable the implementation of the desired change. If the is not the case, a new iteration of the recovery step with the changed components is triggered.

2.5.3 Quality Attribute Driven SAR

The goal of Quality Attribute Driven Software Architecture Reconstruction (QADSAR) [39] is to provide information that enables the analysis of quality attributes of software. The approach is motivated by the fact that business goals incorporate quality attributes. In this context, Stoermer et al. [39] have developed a tool named ARMIN that is described in Section 7.1.

QADSAR uses the notions of quality attribute scenarios and architecture tactics. These are explained in the next paragraph.

Quality attributes are refined into quality attribute scenarios. A quality attribute scenario is a quality attribute requirement of a system. For instance, a system must provide an answer in at most 200 ms. That would create a quality attribute scenario in which a performance requirement is stated. Those scenarios represent the input to a corresponding quality attribute model like a performance model. An architecture tactic can then be chosen by the software architect to accomplish this requirement. In the aforementioned example the architect could choose the tactic *reduce computational overhead*.

Figure 8 taken from Stoermer et al. [39] shows the different steps in QADSAR. The first three steps, namely Scope Identification, Source Model Extraction, and Source Model Abstraction map roughly to the first three common phases described in Section 2.1. The

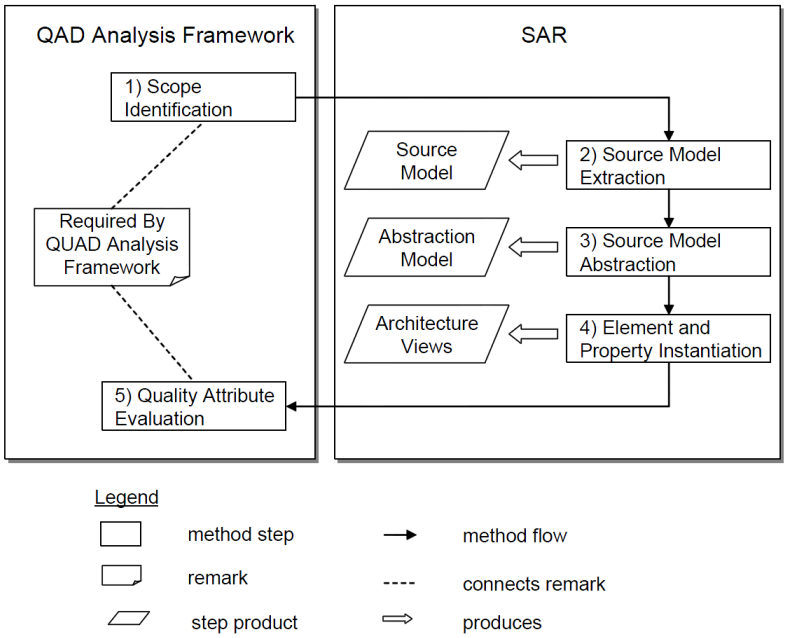

Figure 8: QADSAR steps

only difference is the fact that their scope identification includes the identification of the quality attribute scenario that should be reconstructed and the related architecture tactics. Step four, namely Element and Property instantiation, describes the process of making the entities and relations explicitly denoted as architecture elements with particular properties. The abstraction phase described in Section 2.4 includes this phase. Stoermer et al. introduce a new step named *Quality Attribute Evaluation*. In this step the results of the reconstruction process are evaluated on the basis of the identified quality attribute scenario, quality attribute model, and the possible architecture tactics. For instance, assuming that the scope identification step identified a performance model and the possible tactics to achieve this requirement and the SAR process is performed. If no tactics can be identified in the results, the performance is expected to fail the requirement.

3 Knowledge Discovery Meta-Model

KDM [17] defines different meta-data that play an important role in SAR. KDM maps information about software assets, their associations, and operational environments into one common data interchange format. Then, different analysis tools have a common base for interchanging information. Thereby, the different architecture views, which every analysis tool extracts, can be kept in one meta-model. For this purpose KDM provides various levels of abstraction represented by entities and relations. Section 3.1 provides an overview of the structure and organization of KDM. Afterwards, Section 3.2 presents an example for a KDM-conform file.

3.1 Structure of KDM

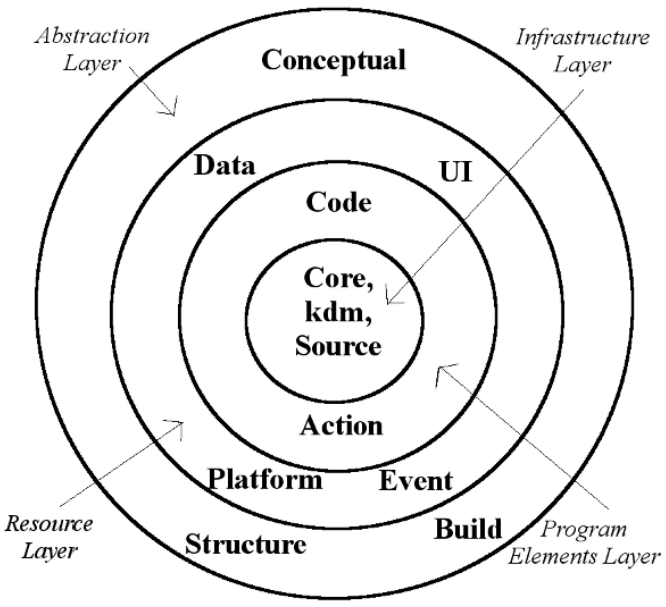

Figure 9: The different layers of KDM

KDM consists of four different layers. These four layers are split into several different packages (see Figure 9 which is based on a Figure from the KDM documentation [17]). Each package, except the core and kdm package, defines one model. Thus, KDM has nine models for representing knowledge about the software architecture. The remainder of the Section describes the different packages of each layer of KDM.

Infrastructure layer

This layer describes the core components of KDM. Every model in other layers inherits directly or indirectly from these components.

- Core package: This package describes the basic meta-classes. For example, the basic classes *KDMEntity* and *KDMRelationship* are defined here. Every element of KDM inherits directly or indirectly from one of the core classes.
- kdm package: The kdm package provides static context shared by all KDM models.
- Source package: This package defines the source model. The source model represents the physical structure of the existing software system. This structure includes the directory structure and files of the file system.

Program layer

The program layer defines a language-independent representation of the existing source code.

- Code package: Elements of programming languages are described in this package, e.g. classes, data types, methods, and variables. Providing a maximum of language independence is the intention of this package. Every case in which this is not possible the source code line is attributed with the dependent programming language descriptor.
- Action package: The behavior and interactions of the instructions among each other are covered here. Function calls and variable assignments are examples for the behavior of a software system.

Resource layer

Higher-level knowledge about the existing software system is represented in this layer.

- Data package: The persistent data aspects of an application are handled in this package.
- UI package: This packages represents the user-interface aspects of the existing software system.
- Event package: In this package a common concept related to event-driven programming is defined.
- Platform package: The artifacts which relate to the runtime platform are handled here.

Abstraction layer

This layer contains even higher-level abstractions about the existing software system than the resource layer.

- Conceptual package: This package is used for representing the business logic and the domain-specific elements.

- Structure package: The logical organization in subsystems, components, and packages is covered here.
- Build package: The engineers view of the software system is represented here. With the build package artifacts and processes in the build process can be described.

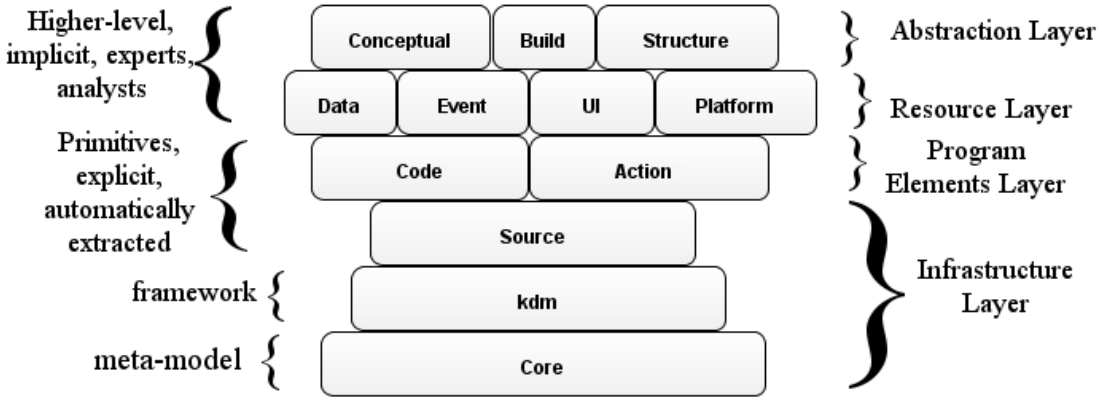

Figure 10: Structure of KDM

Figure 10, basing on a Figure from the KDM documentation [17], displays the structure of KDM with emphasize on the possible level of automatism of creating each package model. At the bottom lie the *core* and *kdm* package that every model inherits from. On a higher level is the *source*, *code*, and *action* package. These packages can be full automatically extracted from the source code. The packages, displayed above them, can only be partially extracted automatically. For these packages human interaction is required to fully represent the information about the system. For example, the information concerning the *conceptual* package is mainly included in the source code but it is nearly impossible to extract all of the business logic automatically because it is only included implicit. Thus, an human has to provide the input.

3.2 Example for a KDM-conform File

```
1 int main(int argc, char* argv[]) {
2 }
```

Listing 5: "KDM simple C example"

Listing 5 shows a typical main procedure of a C program with an empty body. The procedure is stored in the file *simple.c*.

The source code in Listing 5 is converted to a KDM-conform file. The result is depicted in Listing 6, which bases on a *hello world* example by KDM Analytics [2]. At the lines 2 to 6 are the namespace imports. KDM bases on XML Metadata Interchange (XMI).

Therefore, it includes a version of XMI namespace. In addition, the example uses the *code*, *kdm*, and *source* package and therefore, these namespaces must be included as well.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <kdm:Segment xmi:version="2.1"
3  xmlns:xmi="http://www.omg.org/XMI"
4  xmlns:code="http://kdm.omg.org/code"
5  xmlns:kdm="http://kdm.omg.org/kdm"
6  xmlns:source="http://kdm.omg.org/source"
7
8  name="SimpleExample">
9    <model xmi:id="id.0" xmi:type="code:CodeModel" name="SimpleExample">
10     <codeElement xmi:id="id.1" xmi:type="code:CompilationUnit" name="
           simple.c">
11       <codeElement xmi:id="id.2" xmi:type="code:CallableUnit" name="
             main" type="id.4" kind="regular">
12         <source xmi:id="id.3" language="C" snippet="int main(int argc,
               char* argv[]) {}"/>
13         <codeElement xmi:id="id.4" xmi:type="code:Signature" name="main
               ">
14           <source xmi:id="id.5" snippet="int main(int argc, char * argv
                 []);"/>
15           <parameterUnit xmi:id="id.6" name="argc" type="id.12" pos="1"
                 />
16           <parameterUnit xmi:id="id.7" name="argv" type="id.8" pos="2">
17             <codeElement xmi:id="id.8" xmi:type="code:ArrayType">
18               <itemUnit xmi:id="id.9" type="id.11"/>
19             </codeElement>
20           </parameterUnit>
21         </codeElement>
22       </codeElement>
23     </codeElement>
24     <codeElement xmi:id="id.10" xmi:type="code:LanguageUnit">
25       <codeElement xmi:id="id.11" xmi:type="code:StringType" name="char
             *"/>
26       <codeElement xmi:id="id.12" xmi:type="code:IntegerType" name="int
             "/>
27     </codeElement>
28   </model>
29   <model xmi:id="id.13" xmi:type="source:InventoryModel" name="
         SimpleExample">
30     <inventoryElement xmi:id="id.14" xmi:type="source:SourceFile" name=
         "simple.c" language="C"/>
31   </model>
32 </kdm:Segment>
```

Listing 6: "KDM simple example"

In line 9 a code model from the *code* package is begun. After this, the only file *simple.c* is opened. The following line defines a callable unit. This unit is the main procedure from the C example. Then, line 12 defines the original source code of the main procedure. It attributes the source code as C code. This way special behavior of, for instance, variable declarations in different programming languages can be modeled. The next lines 13 to 21 define the signature for the main procedure. The types of the parameters are modeled in lines 24 to 27. Line 29 defines an inventory model from the *source* package. This model represents the source files that are included. In the example there is only the *simple.c* file. This file is represented in line 30.

4 Evaluation of different Interchange Formats for Software Architecture Reconstruction

		KDM	GXL	RSF	FAMIX
C1	**Bases on**	XMI	XML	Tuples	CDIF
C2	**Completeness**	+	+	+	+
C3	**Evolvability**	+	+	+	+
C4	**Flexibility**	+	+	+	+
C5	**Formality**	+	+	+	+
C6	**Included predefined models**	+	−	−	±
C7	**Scalability**	+	+	+	+
C8	**Several levels of abstraction**	+	±	±	±
C9	**Solution reuse**	+	+	−	+
C10	**Static and dynamic dependencies**	+	+	+	+

Legend: +: supported ±: minimal −: not at all

Table 1: Evaluation of different interchange formats

Table 1 summarizes the evaluation of the different common interchange formats for SAR. GXL and RSF are examples from the class of data interchange formats with graph representation. FAMIX is another meta-model based approach to evaluate the functionality of KDM in its own equivalence class.

The legend reads like follows. *Supported* stands for full feature support. *Minimal* stands for an only partially supported feature. *Not at all* stands for a feature that is not supported at all.

The evaluation was conducted with KDM in version 1.1 [17], GXL in version 1.1 [24, 48, 49], RSF in version 5.4.4 (handbook's version) [19, 46, 50, 51], and FAMIX in version 2.0 [9, 11, 42].

Section 4.1 defines the different assessment criteria. Then, Section 4.2 to 4.5 describe the results for the different formats in detail.

4.1 Assessment criteria

St-Denis et al. [37] propose the following assessment criteria for evaluation of interchange formats. However, some aspects were omitted and added. This has been done to scope the evaluation for SAR. The omitted criteria are transparency, simplicity, neutrality, popularity, metamodel identity, legibility, and integrity. The added aspects are bases on, included predefined models, several levels of abstraction, and static and dynamic dependencies.

C1 Bases on The underlying format is an important criterion and affects implicitly to other criteria like scalability.

C2 Completeness All necessary components and algorithms should be specified. The specification must provide explicit and unambiguous guidelines for the user. Hence, the potential of incompatibility is reduced.

C3 Evolvability New requirements should be easy to implement. Thus, the interchange format must provide an easy way to adapt new requirements.

C4 Flexibility The interchange format should be so flexible that different models, tool-specific data, and language systems can be included.

C5 Formality The specification of the format should be formal and well-defined. Resulting in a well understanding and a non conflicting interpretation. Hence, incompatibility and information corruption is avoided.

C6 Included predefined models For easier interoperability the specification should include predefined models for the representation of data. If there are no predefined models included the format fails this criterion. If only one model is included, the format fulfills the requirement only partially.

C7 Scalability The format should handle small and huge systems. Therefore, the format must be able to deal with a great amount of information.

C8 Several levels of abstraction The interchange format should provide different levels of abstraction. On a lower level of abstraction the source model should be contained and at a higher stage the subsystem structure should be contained, for instance. If only one level of abstraction is possible, the format fails the requirement. If it is possible to define several levels of abstraction by extending the format, the criterion is partially fulfilled.

C9 Solution reuse Solution reuse consists of adapting and integrating standards, tools, and other reusable concepts or mechanisms. The format should reuse these proven technologies for a reduction of the error probability. The solution reuse criterion does not include reused solutions which result from the base format.

C10 Static and dynamic dependencies Static and dynamic data should be representable in the format.

4.2 Knowledge Discovery Meta-Model

KDM *bases on (C1)* XMI. XMI is a standard by the Object Management Group (OMG) which combines the meta-meta-model (MOF) and a textual format, namely XML. For KDM a documentation with a description, semantics, and special constraints of its elements is available from its website [17]. Hence, KDM fulfills the *completeness (C2)* requirement. Generic extensions of the existing models and the possibility of defining new models make KDM *evolvable (C3)*. New models can be implemented on the basis of the core and kdm package and tool specific data can be added as annotations, for instance. Thus, KDM is *flexible (C4)*. The specification defines the semantic and constraints for the KDM elements. Therefore, KDM fulfills the *formality (C5)* criterion. There are 9

predefined models included (C6) in KDM. These models are described in Section 3. XMI is a verbose format but KDM *scales up (C7)* that is shown by KADis which can analyze systems with millions of lines of code. The models included in KDM provide *different levels of abstraction (C8)*. For example, the code model contains classes and the structure model can contain the structure of components of the software system. KDM *reuses (C9)* ISO norms for primitive types, for instance. The *static and dynamic dependencies (C10)* can be expressed, for example, by the *code* and *action* package of KDM.

4.3 Graph eXchange Language

GXL *bases on (C1)* the XML format. XML was chosen by the developers of GXL instead of XMI to have a less verbose base format. Since GXL is specified comprehensively by a DTD, it fulfills the *completeness (C2)* requirement. Graphs can be easily extended and new requirements can be implemented by changing the semantic schema. In GXL not only the graph is represented but its underlying schema definition is exchanged as a metaschema with the graph representation. Hence, GXL provides *evolvability (C3)*. Through changing the metaschema, tools can provide their own specific data and models. Thus, GXL is *flexible (C4)*. The DTD for GXL provides a *formal (C5)* and well-defined specification for the format. GXL has no *included predefined models (C6)*. The format defines only a graph representation. However, programs can define their own models with a metaschema but the specification has none metaschema included. TA, GraX and PROGRES were merged to create GXL. These formats have been proven in several large software analyses. For instance, TA was successfully used at analyzing the Linux kernel and GCC C++ compiler. Therefore, GXL *scales up (C7)* and can handle large software systems too. Nearly every abstraction level can be represented as a graph. Only the underlying metaschema changes at every abstraction level. This metaschema can be manipulated in GXL. Hence, GXL provides the possibility of *several levels of abstractions (C8)* but none is included. GXL evolved from different formats. Therefore, it *reuses (C9)* these solutions. *Static and dynamic dependencies (C10)* can be represented by graphs. Hence, GXL can represent them by altering the metaschema.

4.4 Rigi Standard Format

RSF *bases on (C1)* tuples. The tuple rules are described in Section 2.3.3. A specification and examples for RSF are available from its handbook. Hence, RSF fulfills the *completeness (C2)* requirement. Like in GXL, graphs can be easily extended and new requirements can be implemented by changing the underlying semantic schema. Therefore, RSF is *flexible (C3)*. With the tuples in RSF it is possible to construct a graph. Graphs are *evolvable (C4)* for new requirements. A *formal (C5)* definition of RSF is available from the Rigi wiki [46]. RSF only defines a graph representation and no concrete models. Hence, there are no *included predefined models (C6)*. Rigi utilizes RSF and has been proven in large scale systems of around 5 million lines of code. Therefore, RSF *scales up (C7)*. Through

different graphs with a different underlying metaschema *different levels of abstraction (C8)* can be achieved but only one is included. No solution reuse was described in the papers and no solution reuse was obvious except the underlying base format. Therefore, RSF does not *reuse solutions (C9)*. Graphs can represent *static and dynamic dependencies (C10)*. Hence, RSF fulfills this requirement.

4.5 FAMIX Meta-Model

FAMIX *bases on (C1)* CDIF. CDIF is an industrial standard that was chosen for FAMIX due to its extensibility and standard plain text encoding. FAMIX is defined by a specification with a description for each element. Therefore, it fulfills the *complete (C2)* criterion. New requirements to the format can be implemented by inheriting from the abstract classes defined in FAMIX. Hence, FAMIX is *evolvable (C3)*. Tool specific data and own models can be included in FAMIX by extending the abstract base classes. Thus, FAMIX is *flexible (C4)*. FAMIX is *formal (C5)* and well defined by its specification which describes the elements, their methods, and their attributes. A *predefined model (C6)* for the representation of entities like methods, classes, or packages is contained in FAMIX. However, this is the only one. Moose uses FAMIX for its data storing and Moose has provided adequate results in SAR. Hence, FAMIX fulfills the *scalability (C7)* criterion. *Several levels of abstraction (C8)* can be implemented in FAMIX. However, this would require an extension of the FAMIX model. FAMIX *reuses (C9)* the common entities of object orientation to define its abstract model. *Static and dynamic dependencies (C10)* can be represented in FAMIX. The creators of FAMIX have focused on the fact that a SAR format needs *invocation* and *access* entities.

5 Development of KADis

KADis is a new tool, which I developed within the scope of this thesis, to enable SAR with the usage of KDM. It is implemented as an Eclipse-RCP application. The source code of KADis and the tool itself can be downloaded from its website [14]. The future work for further versions of KADis is described in Section 8.2. Section 5.1 and 5.2 present an overview of the program by listing the features and the design of KADis. The last Section 5.3 shows the different activities that are performed by KADis in its main feature accomplishing the SAR process.

5.1 Features

KADis enables the reconstruction of software architectures and uses Source Navigator NG [36] for the data extraction phase. Source Navigator NG is an open-source project and thus enables further development and extension of the included parsers. In the first version KADis only supports Java. However, interfaces for further implementation of other languages are contained. Basically, classes, packages, and associations are reconstructed by using the code- and structure package of KDM. The output of the SAR process in KADis is an XML file in a KDM-conform format (see Section 3.2 for an example).

Appendix D contains the functional specification of KADis. It includes further information about the features and other functional details of the program.

5.2 Design

The first version of KADis provides a basis for a powerful SAR tool utilizing the KDM. Therefore, the main intention of the design is focused onto extensibility and exchangeability of the created components. Thus, KADis implements the Model-View-Controller (MVC) architectural pattern, which separates the view, controller, and model from each other. For future versions it is planed that KADis should act in a self developed framework. Therefore, the view has to be independent.

For more information about the design of KADis refer to the design specification in Appendix E.

5.3 Activities in KADis' SAR process

KADis covers the SAR phases which were defined in Section 2. Figure 11 shows the different activities that are being performed by KADis in detail. The first activity is *scope identification*. This activity is carried out by the user by selecting the files and folders that the following SAR process should use. After the SAR process is started by the user, KADis starts the parsing of the passed files and folders in the *data extraction* activity with special focus on the information needed by the *code* and *source* package of KDM. KADis uses Source Navigator NG for this task. Then, the results of the parsing

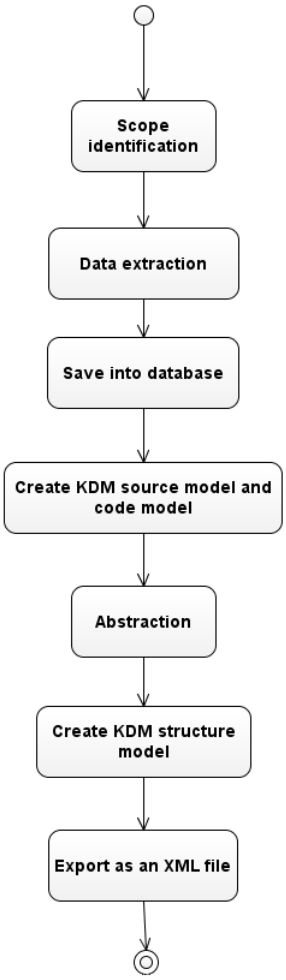

Figure 11: Activities in KADis' SAR process

are saved into a database in the *save into database* activity. The developed database schema is based on the KDM specification. After Source Navigator NG has finished, a Java Abstract Syntax Tree (AST) is used to verify and complete the results. Then, a *code* and *source* model instance of KDM are created from the database. Subsequently, KADis abstracts the information gathered in the *data extraction* activity to create a *structure* model instance of KDM. Finally, the program exports the three created model instances as one KDM-conform XML file.

6 Evaluation of KADis

This Section evaluates the functionality of KADis. The completeness evaluation is conducted with JPetStore 5.0 and is described in Section 6.1. The following Section 6.2 shows the performance evaluation of KADis.

6.1 Completeness Evaluation with JPetStore

JPetStore 5.0 [25] is a web store for pets published by iBatis. The program is a wide used program for evaluation purposes. The generated output for JPetStore can be downloaded from the KADis website [14].

KDM is a verbose format. A hello world example in KADis consists of 50 lines as KDM format. Thus, only an evaluation summary in table 2 is shown for JPetStore. The elements that were detected by KADis were measured by counting the occurrence of the corresponding KDM elements type with a text tool. The words, which were searched for each criterion, are contained in each criterion definition. The total number of the elements present in JPetStore was counted manually and Appendix C shows a detailed table for the calculation of the total number in the files. Section 6.1.1 defines the assessment criteria and Section 6.1.2 discusses the results.

		Found	Total	Percentage
E1	**Directories**	41	41	100 %
E2	**Files**	171	171	100 %
E3	**Packages**	6	6	100 %
E4	**Classes**	49	49	100 %
E5	**Imports**	220	220	100 %
E6	**Inheritances**	42	42	100 %
E7	**Instance Variables**	150	150	100 %
E8	**Methods**	421	421	100 %
E9	**Local Variables**	273	273	100 %

Table 2: Detected elements by KADis and total elements of JPetStore

6.1.1 Assessment Criteria

KADis' output contains the three models from the source, code, and structure package. These models have to be evaluated. The source package defines the inventory model which represents the physical structure of the source code. Directories and files are mostly contained in this model. For this purpose, E1 and E2 are chosen. The code model represents the source code. KADis 1.0 only supports Java and is therefore only applicable for object-oriented software. E3 to E9 define the main elements of an object-oriented language. The structure model generated by KADis contains a textual description of the

package dependencies in the system. For the package diagram, packages (E3) and imports (E5) have to be correctly detected.

E1 Directories Directories are part of the physical organization of the source code. KDM represents the directory structure by recursion. The first directory entry defines the absolute path to the root source folder. The child directory entry defines the name of the root source folder. The following directories are always child of these and always define the relative location of the directory container. A directory in a KDM-file is of the type *source:Directory*.

E2 Files Files are containers for different data. The type of a file in KDM can be *source:SourceFile*, *source:ResourceDescription*, *source:Image*, *source:ExecutableFile*, *source:BinaryFile*, or *source:Configuration*.

E3 Packages Packages are the Java specific representation of a module. Only the packages that contain source files are counted. In KDM the type of a package is *code:Package*.

E4 Classes Classes can be normal classes, interfaces, or enumerations here. Inner classes are counted as a normal class. The corresponding types in KDM are *code:ClassUnit*, *code:InterfaceUnit*, and *code:EnumeratedType*.

E5 Imports Imports define the usage of other classes and packages. The imports are only counted once for a file. The type for imports in KDM is *code:Imports*.

E6 Inheritances An inheritance relation can be of two types, namely extends and implements. These types are mapped by KDM to *code:Extends* and *code:Implements*.

E7 Instance Variables Instance variables are the global variables of a class. In KDM variables are of the type *code:StorableUnit* and an instance variable is of the type *global* or *static*.

E8 Methods In this context methods mean constructors and normal method declarations. KDM defines the type *code:MethodUnit* for methods.

E9 Local Variables Local variables always belong to one method and are counted for each declaration. Though anonymous variables, for example *new Integer*, are not counted. KDM maps local variables to the type *code:StorableUnit* with the type *local*.

6.1.2 Discussion

The directory criterion definition already stated that KDM includes the path and name of the root source folder. These two directories entries were subtracted from the result of the search for *source:Directory* in the KDM-file.

The results show that KADis detects the basic elements of Java. Though looking at the source code reveals different shortcomings which result from using Source Navigator NG. Source Navigator NG fails to detect inner classes correctly and simply adds two classes with the main class name. This can result in mismatching modifiers as happened in the class *ProductSqlMapDao* which is a public class but is detected as a public static class,

for instance. Another shortcoming is the detection of package private modifiers. Source Navigator NG detects this modifier as a private modifier. In future versions of KADis these shortcomings shall be resolved.

An evaluation with one program is not necessarily generalizable. Though, the evaluation shows that KADis detects a large part of main elements of Java.

6.2 Performance Evaluation

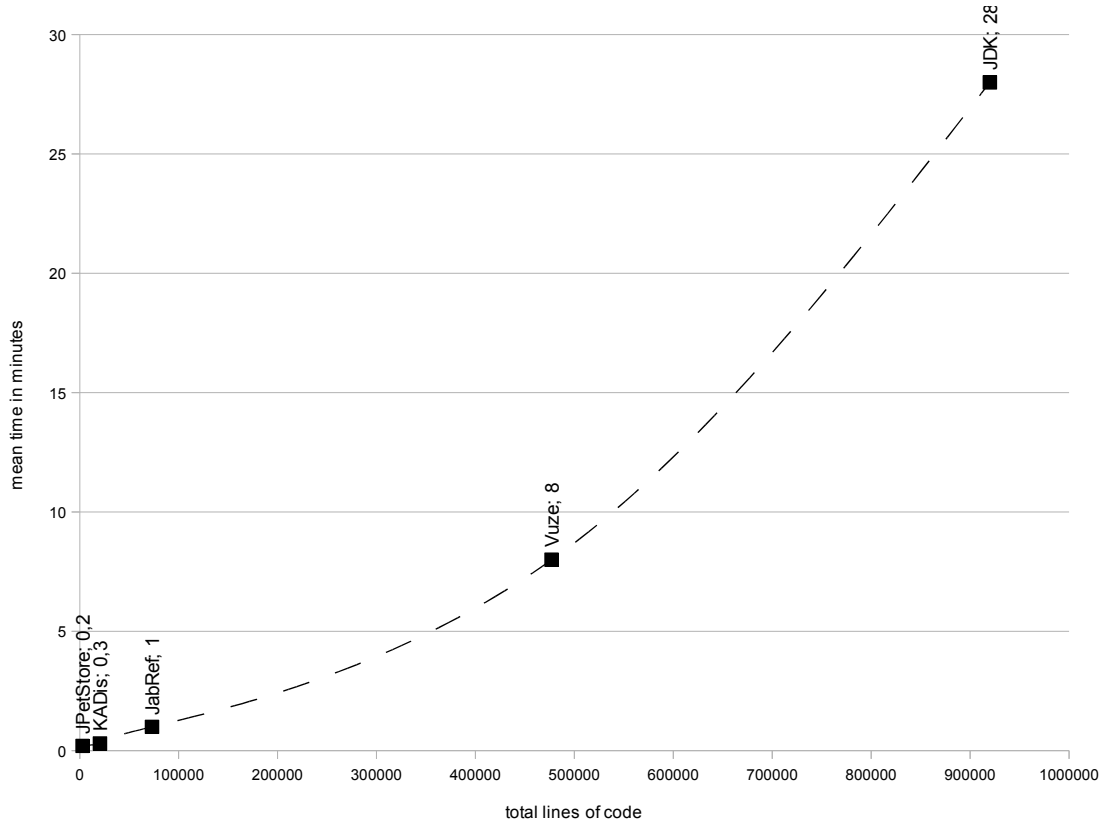

Figure 12: Performance evaluation

In the majority of cases SAR must provide a fast output despite handling a big amount of data. Therefore, KADis was optimized for generating the output as fast as possible by tuning the used database schema and the database itself, and by implementing different threads in the *linking ids* step, which links the reference from an KDM element to another KDM element. In addition, optimizations for memory usage were done by using a relational database without an object-relational mapper to control the data that the RAM has to hold.

The evaluation was done on an Intel Core 2 Duo E7200 2.53 GHz and 4 GB RAM and a maximum heap size for the JVM of 1536 MB. The versions of the used programs are JDK [32] 1.6 Update 18 (the contained src.zip), Vuze [47] 4.3.1.2, JabRef [41] SVN

revision 3161 of the main program, KADis 1.0 [14] and JPetStore 5.0. Source code written in other languages than Java was removed before the reconstruction was accomplished.

Figure 12 shows the performance of KADis processing the aforementioned software. The total lines of code (LOC) on the x-axis were measured by Metrics [34]. LOC shall only be a hint for the size of the project here. The mean time in minutes were taken from the displayed time in KADis after the reconstruction has finished. This time is the difference between the click on the start reconstruction button and after the KDM-file is written to the hard disk. To calculate the mean time, the SAR process for each program was conducted 3 times and to guarantee the same conditions KADis was restarted after each SAR process. The dashed curve defines an estimated mean time. The values behind the items are the concrete mean time values in minutes for the item.

The curve in Figure 12 lets assume that KADis' output generation is above linear time but within square time. An explanation can be found in the underlying algorithms. The link algorithms in KADis, for instance the former mentioned id linking step, have $O(n*m)$ time where n stands for the number of elements that have to be linked from and m for the number of available elements in the KDM instance. These algorithms, which form the bottleneck here, have to be optimized in future versions by search trees, for example.

7 Related Tools in Software Architecture Reconstruction

There are many tools available that perform SAR. Section 7.1 to 7.3 list only a few tools that were mentioned in other Sections of the thesis. An exception is made for MoDisco, which is a framework which uses KDM to describe software architectures, and that is covered in greater detail in Section 7.4.

7.1 Architecture Reconstruction and MINing

Architecture Reconstruction and MINing (ARMIN) [39] is a tool developed by the Software Engineering Institute and Robert Bosch Corporation. It uses RSF for importing files and provides configurable modeling and visualization. ARMIN is a successor of the Dali architecture reconstruction workbench [26] and realizes the QADSAR approach, which is described in Section 2.5.3.

7.2 Rigi

Rigi [19] is a semi automatic reverse engineering environment, which was developed at the University of Victoria. It consists of several tools. These are parsers, command-line utilities, and an interactive graph editor, which is the core of the system and is called *rigiedit*. Graph models are stored and retrieved by this tool. It is programmable by using the scripting language Tcl, which is a library of scripts supplied for performing common reverse engineering tasks. The data is represented in its own format named RSF.

7.3 Moose

Moose [20] is a language-independent environment for reverse and re-engineering software systems. The project started at the Software Composition Group in 1997. The tool is an open source software and utilizes a Smalltalk implementation of FAMIX. Moose offers a set of different services. The tool includes a common meta-model, visualization, and a model repository, for instance. Moose was developed in the context of FAMOOS, which was an European project whose goal was to support the evolution of first generation object-oriented software towards object-oriented frameworks.

7.4 MoDisco

MoDisco [40] provides a framework to develop model-driven tools in different scenarios of software modernization. These scenarios are mostly *quality assurance*, *documentation*, *improvement*, and *migration*. In the *quality assurance* scenario it shall be verified whether an existing system fulfills the quality requirements. The *documentation* scenario requires extracting information from a system to support the process of understanding the system.

The *improvement* scenario transforms an existing system to integrate, e.g., design patterns. Finally, the *migration* scenario transforms an existing system to change, for instance, a component.

For these purposes MoDisco provides different supporting components. Firstly, MoDisco offers different meta-models to describe the existing system. Secondly, it includes *discoverers* for the automatic extraction of these models. Finally, the framework includes generic tools to understand and transform the created models.

8 Conclusion and Future Work

The remainder of the Section concludes the main aspects of the thesis in Section 8.1 and defines the future work in Section 8.2.

8.1 Conclusion

Different approaches to SAR were shown in Section 2. These approaches were categorized into a new categorization, which focuses on the basic phases of SAR and therewith the understanding of SAR. The common data interchange formats for SAR were compared in Section 4 to evaluate the suitability of KDM. The result of this evaluation is the fact that KDM meets all defined requirements and is more formal defined than the existing FAMIX format, which is widely used in similar contexts. Therefore, enabling better interoperability between different tools. KADis, a new tool for SAR, was developed to evaluate the practicability of KDM. The evaluation in Section 6 has shown that KADis generates the expected output. Hence, KDM was practically utilized and thus KDM has proven useful in SAR.

8.2 Future Work

Most of the future work lies in extending KADis. KDM is a huge specification. Therefore, KADis only implements the *core*, *source*, *kdm*, *code*, and *structure* package at this time. The other packages, namely *action*, *data*, *event*, *platform*, *ui*, *conceptual*, and *build* package, have to be integrated to fully support KDM.

SAR processes are mostly iterative and incremental and performed semi-automatically. Different points come up with this circumstance. Firstly, the user has to interact with the SAR program. Hence, a graphical domain-specific language, that represents KDM elements, has to be developed. With this graphical representation the reverse engineer can make changes to the software system, for instance. Secondly, the reconstructed KDM-conform files have to be merged at the end of each iteration. Therefore, KADis has to provide a merge system that allows the user to decide whether or not to adopt the made changes. Thirdly, an easy scripting language for presenting only special points of interest, for instance only names which start with *add*, of the architecture has to be developed and integrated into KADis. In addition, the GUI can support the process of writing such queries. Lastly, the computation of an iteration should complete as fast as possible. Therefore, further threads should be implemented in the data gathering and abstraction phase to meet this requirement, for instance.

Source Navigator NG has different shortcomings in parsing, since it only supports Java 1.0. I have added elements like enumerations by the usage of a Java AST implementation from the Eclipse package *org.eclipse.jdt.core.dom*. Although, some elements are still not being detected correctly. These elements are inner classes, generics, overloaded methods, and annotations in methods. In future versions these wrongly parsed elements must be

detected and then be resolved by, for example, the Java AST. An alternative would be that the Java AST should be the only parser for Java.

Many huge software systems are written in more than one programming language. Thus, other programming languages than Java have to be supported in KADis, too. This raises the problem of interoperability between different languages. Many of the old software systems are written in procedural languages, like COBOL. Therefore, an abstraction mechanism that finds classes has to be implemented in KADis.

For a large system there may be more than one reverse engineer involved in SAR. Therefore, the project system must be extended to save all relevant data, e.g. source files and resource files, and to provide the possibility of an easy interchange. In a system with million lines of code and other resource files, like build scripts, the project file can be very large. The problem of finding an efficient way to save the data by compression or dummy files, for instance, must be solved.

References

[1] Approximation Algorithms for NP-Hard Problems. *SIGACT News*, 28(2):40–52, 1997. ISSN 0163-5700.

[2] KDM Analytics. KDM examples. `http://kdmanalytics.com/kdmexamples`, 2010-02-16.

[3] Ivan T. Bowman and Richard C. Holt. Reconstructing Ownership Architectures To Help Understand Software Systems. Technical report, University of Waterloo, 1999.

[4] Gerardo Canfora and Aniello Cimitile. Software Maintenance. Technical report, University of Sannio, 2000.

[5] Gerardo Canfora and Massimuliano Di Penta. Frontiers of Reverse Engineering: a Conceptual Model. Technical report, University of Sannio, Benevento, Italy, 2008.

[6] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architecture: Views and Beyond*. Addison-Wesley, 2002.

[7] Reidar Conradi. Software Engineering mini glossary. `http://www.idi.ntnu.no/grupper/su/publ/ese/se-defs.html`, 2010-02-16.

[8] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why FAMIX - Shortcomings of UML for Round-trip Engineering. Technical report, University of Berne, 1999.

[9] Serge Demeyer, Sander Tichelaar, and Patrick Steyaert. FAMIX 2.0 The FAMOOS Information Exchange Model. Technical report, University of Berne, 1999.

[10] Stéphane Ducasse and Damien Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. Technical report, Centre de Recherche Inria Lille, 2009.

[11] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. MOOSE: an Extensible Language-Independent Environment for Reengineer Object-Oriented Systems. Technical report, University of Berne, 2000.

[12] Jürgen Ebert, Bernt Kullbach, and Andreas Winter. GraX - An Interchange Format for Reenginieering Tools. In *in Sixth Working Conference on Reverse Engineering. IEEE Computer Society, Los Alamitos*, pages 89–98. IEEE Computer Society Press, 1999.

[13] Loe Feijs, René Krikhaar, and Rob van Ommering. A Relational Approach to Support Software Architecture Analysis. *Software, Practice Experience*, 28(4):371–400, 1998.

[14] Florian Fittkau. KADis. `http://sourceforge.net/projects/kadis/`, 2010-02-16.

[15] Rony G. Flatscher. Metamodeling in EIA/CDIF—meta-metamodel and metamodels. *ACM Trans. Model. Comput. Simul.*, 12(4):322–342, 2002. ISSN 1049-3301.

[16] Bernd Krieg-Brückner's Group. uDraw(Graph). `http://www.informatik.uni-bremen.de/uDrawGraph`, 2010-02-16.

[17] Object Management Group. Knowledge Discovery Meta-Model. `http://www.omg.org/spec/KDM/1.1/`, 2010-02-16.

[18] Object Management Group. XML Metadata Interchange. `http://www.omg.org/spec/XMI/2.1.1/`, 2010-02-16.

[19] Rigi Group. Rigi Website. `http://www.rigi.csc.uvic.ca`, 2010-02-16.

[20] Software Composition Group. Moose Homepage. `http://www.moosetechnology.org/`, 2010-02-16.

[21] George Yanbing Guo, Joanne M. Atlee, and Rick Kazman. A Software Architecture Reconstruction Method. Technical report, University of Waterloo and Carnegie Mellon University, Pittsburgh, 1999.

[22] Michael Himsolt. GML: A portable Graph File Format. Technical report, University of Passau, 1997.

[23] Richard C. Holt. An Introduction to TA: The Tuple-Attribute Language. Technical report, University of Waterloo, 1997.

[24] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Toward a Standard Exchange Format. Technical report, University of Waterloo and University of Koblenz-Landau and University Bw, Muenchen, 2000.

[25] iBATIS team. JPetstore. `http://mirror.synyx.de/apache/ibatis/binaries/ibatis.java/JPetStore-5.0.zip`, 2010-02-16.

[26] R. Kazman and S.J. Carrière. Playing Detective: Reconstructing Software Architecture from Available Evidence. Technical report, 1999.

[27] Rick Kazman, Liam O'Brien, and Chris Verhoef. Architecture Reconstruction Guidelines, Third Edition. Technical report, Carnegie Mellon University, Pittsburgh, November 2003.

[28] Rainer Koschke. Architecture Reconstruction, Tutorial on Reverse Engineering to the Architectural Level. Technical report, University of Bremen, 2009.

[29] Giuseppe Antonio Di Lucca, Anna Rita Fasolino, and Ugo de Carlini. Recovering Use Case Models from Object-Oriented Code: A Thread-Based Approach. *Reverse Engineering, Working Conference on*, 0:108, 2000. ISSN 1095-1350.

[30] Nenad Medvidovic and Vladimir Jakobac. Using software evolution to focus architectural recovery. Technical report, University of Southern California, 2006.

[31] Gail Murphy, David Notkin, and Kevin Sullivan. Extending and Managing Software Reflexion Models. Technical report, University of British Columbia and University of Washington and University of Virginia, 1997.

[32] Oracle. JDK website. `http://java.sun.com/javase/downloads/index.jsp`, 2010-02-16.

[33] Matthias Rohr, Andre van Hoorn, Jasminka Matevska, Nils Sommer, Lena Stoever, Simon Giesecke, and Wilhelm Hasselbring. Kieker. `http://kieker.sourceforge.net/`, 2010-02-16.

[34] Frank Sauer. Metrics website. `http://sourceforge.net/projects/metrics/`, 2010-02-16.

[35] Andy Schürr, Andreas Winter, and Albert Zündorf. The Progres Approach: Language And Environment. Technical report, University of München and University of Aachen and University of Paderborn, 1999.

[36] sourcenav NG development group. Source Navigator NG. `http://sourcenav.berlios.de/`, 2010-02-16.

[37] Guy St-Denis, Reinhard Schauer, and Rudolf K. Keller. Selecting a Model Interchange Format, The Spool Case Study. Technical report, University of Montreal, 2000.

[38] Christoph Stoermer, Anthony Rowe, Liam O'Brien, and Chris Verhoef. Model-centric software architecture reconstruction. Technical report, Robert Bosch Corporation, Pittsburgh and Carnegie Mellon University, Pittsburgh and Software Engineering Institute, Pittsburgh and Free University of Amsterdam, 2000.

[39] Christoph Stoermer, Liam O'Brien, and Chris Verhoef. Moving Towards Quality Attribute Driven Software Architecture Reconstruction. Technical report, University of Bremen, 2009.

[40] AtlanMod Team. MoDisco. `http://wiki.eclipse.org/MoDisco`, 2010-02-16.

[41] JabRef team. JabRef website. `http://sourceforge.net/projects/jabref/`, 2010-02-16.

[42] Sander Tichelaar, Stéphane Ducasse, and Serge Demeyer. FAMIX and XMI. Technical report, University of Berne and University of Antwerp, 2000.

[43] S. Tilley and D. B. Smith. Perspective on Legacy System Reengineering. Technical report, Carnegie Mellon University, 1995.

[44] Qiang Tu and Michael W. Godfrey. The Build-Time Software Architecture View. Technical report, University of Waterloo, 2001.

[45] Arie van Deursen, Christine Hofmeister, Rainer Koschke, Leon Moonen, and Claudio Riva. Symphony: View-Driven Software Architecture Reconstruction. Technical report, University of Technol., Netherlands, 2004.

[46] Eelco Visser. Rigi RSF. `http://www.program-transformation.org/Transform/RigiRSF`, 2010-02-16.

[47] Vuze Inc. Vuze website. `http://www.vuze.com/`, 2010-02-16.

[48] Andreas Winter. Exchanging Graphs with GXL. Technical report, University of Koblenz-Landau, 2002.

[49] Andreas Winter, Bernt Kullbach, and Volker Riediger. An Overview of the GXL Graph Exchange Language. Technical report, University of Waterloo and University of Koblenz-Landau and University Bw, Muenchen, 2002.

[50] Kenny Wong. Rigi User's Manual. Technical report, University of Victoria, 1998.

[51] Kenny Wong, Scott R. Tilley, Hausi A. Mueller, and Margaret-Anne D. Storey. Structural Redocumentation: A Case Study. Technical report, University of Victoria, January 1995.

Appendices

A Acronyms

AST
Abstract Syntax Tree

DTD
Document Type Definition

FAMIX
FAMOOS Information Exchange Model

GXL
Graph eXchange Language

KADis
KDM Architecture Discoverer

KDM
Knowledge Discovery Meta-Model

QADSAR
Quality Attribute Driven Software Architecture Reconstruction

RSF
Rigi Standard Format

SAR
Software Architecture Reconstruction

TA
Tuple Attribute Language

UML
Unified Modeling Language

XMI
XML Metadata Interchange

B Glossary

Software architecture

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. [IEEE Std 1471-2000]

Software artifact

Any piece of software [..] developed and used during software development and maintenance. Examples are requirements specifications, architecture and design models, source and executable code (programs), configuration directives, test data, test scripts, process models, project plans, various documentation etc. [taken from "Software Engineering mini glossary" [7]]

View

A view is a representation of a whole system from the perspective of a related set of concerns. [ISO/IEC 42010:2007]

Viewpoint

A viewpoint is an abstraction that yields a specification of the whole system restricted to a particular set of concerns. [IEEE Std 1471-2000]

C JPetStore elements

File name	Pack	Cla	Imp	Inh	InstV	Met	LocV
Account.java	domain	1	1	1	18	36	0
BeanTest.java	domain	1	11	1	1	2	9
Cart.java	domain	1	5	1	2	10	11
CartItem.java	domain	1	2	1	4	9	0
Category.java	domain	1	1	1	3	7	0
DomainFixture.java	domain	1	2	0	0	2	4
Item.java	domain	1	2	1	13	27	0
LineItem.java	domain	1	2	1	7	16	0
Order.java	domain	1	6	1	27	57	3
Product.java	domain	1	1	1	4	9	0
Sequence.java	domain	1	1	1	2	6	0
AccountDaoTest.java	persistence	1	3	1	1	4	5
BasePersistenceTest.java	persistence	1	2	1	1	1	0
CategoryDaoTest.java	persistence	1	1	1	1	2	0
DaoConfig.java	persistence	1	8	0	2	3	8
ItemDaoTest.java	persistence	1	5	1	4	5	3
OrderDaoTest.java	persistence	1	4	1	3	2	5
PersistenceFixture.java	persistence	1	6	0	5	2	3
ProductDaoTest.java	persistence	1	1	1	1	3	0
SequenceDaoTest.java	persistence	1	1	1	1	1	4
AccountDao.java	iface	1	1	0	0	4	0
CategoryDao.java	iface	1	2	0	0	2	0
ItemDao.java	iface	1	3	0	0	4	0
OrderDao.java	iface	1	2	0	0	3	0
ProductDao.java	iface	1	2	0	0	3	0
SequenceDao.java	iface	1	0	0	0	1	0
AccountSqlMapDao.java	sqlmapdao	1	3	2	0	5	1
BaseSqlMapDao.java	sqlmapdao	1	2	1	1	1	0
CategorySqlMapDao.java	sqlmapdao	1	4	2	0	3	0
ItemSqlMapDao.java	sqlmapdao	1	8	2	0	5	7
OrderSqlMapDao.java	sqlmapdao	1	5	2	0	4	3
ProductSqlMapDao.java	sqlmapdao	2	7	2	1	6	2
SequenceSqlMapDao.java	sqlmapdao	1	4	2	0	2	2
AbstractBean.java	presentation	1	2	1	5	1	0
AccountBean.java	presentation	1	8	1	10	28	3
AccountBeanTest.java	presentation	1	8	1	0	11	38

CartBean.java	presentation	1	7	1	4	14	8
CartBeanTest.java	presentation	1	7	1	0	7	21
CatalogBean.java	presentation	1	5	1	12	31	0
CatalogBeanTest.java	presentation	1	8	1	0	8	18
OrderBean.java	presentation	1	9	1	9	22	10
OrderBeanTest.java	presentation	1	12	1	0	14	65
AccountService.java	service	1	4	0	1	6	1
AccountServiceTest.java	service	1	5	1	0	4	8
CatalogService.java	service	1	10	0	3	10	1
CatalogServiceTest.java	service	1	10	1	0	8	16
OrderService.java	service	1	8	0	4	6	2
OrderServiceTest.java	service	1	9	1	0	4	12
Overall	6	49	220	42	150	421	273

Table 3: Elements of JPetStore

Legend for table 3

File name: The file name which is unique in JPetStore.

Pack: Pack stands for package declaration. To count the distinct packages the package declaration name is displayed in the table.

Cla: Classes, interface, and enumerations are counted in this column.

Imp: Imp are all import statements included in the file.

Inh: Inh stands for the inheritances declared by implement and extend statements.

InstV: InstV are the instance variables in the file.

Met: The methods in the file.

LocV: LocV are the local variables in the file. Variable declarations in for statements are not counted as local variables because they have a special purpose for indexing elements.

D Functional Specification

Functional Specification:

KDM Architecture Discoverer (KADis)

State: 2010-02-15
Version: 1.0

Author:

Florian Fittkau

Contents

1 Purpose **1**
1.1 Required features . . . 1
1.2 Desired features . . . 1
1.3 Optional features . . . 2

2 Scope **3**
2.1 Application range . . . 3
2.2 Target group . . . 3

3 Minimum Requirements **4**
3.1 Software . . . 4
3.2 Hardware . . . 4

4 Product Overview **5**

5 Actors **6**

6 Product Functions **7**
6.1 User Use Cases . . . 7
6.1.1 Manage projects . . . 7
6.1.2 Manage data for reconstruction . . . 8
6.1.3 Set output folder for KDM-file . . . 9
6.1.4 Perform architecture reconstruction . . . 10
6.1.5 Filter log messages . . . 11
6.2 System Use Cases . . . 12
6.2.1 Parse artifacts . . . 12
6.2.2 Get results from external parser . . . 13

7 Product Data **14**
7.1 Projects . . . 14
7.2 KDM-files . . . 14

8 Product Performance **15**

9 Graphical User Interface **16**
9.1 Main window . . . 16
9.2 Dialogues . . . 17
9.3 Layout . . . 17
9.4 Eclipse integration . . . 17

10 Software System's Attributes **18**

11 Test Cases **19**
11.1 Hello World program 19
11.2 More complex test program 19
11.3 JPetStore 19

12 Development Environment **20**
12.1 Software 20
12.2 Hardware 20

References **21**

A Acronyms **i**

B Glossary **ii**

1 Purpose

KDM Architecture Discoverer (KADis) is an Eclipse-RCP application, which enables the user to reconstruct the architecture of a given program on basis of its source code. Section 1.1 to 1.3 outline the features of KADis. For a definition of the different feature types refer to the glossary.

1.1 Required features

- Enabling software architecture reconstruction from Java source code and other system artifacts. The reconstructed architecture is saved in a Knowledge Discovery Meta-Model (KDM)-conform format into a file.
- Using Source Navigator NG [4] for parsing the given source code.
- The Java parser in Source Navigator NG understands Java 1.0 and upwards. If there are needed symbols that can not be parsed and are essential, they will be added by extending the Java parser.
- Basically classes, packages, and associations will be reconstructed by using code- and structure package from KDM. For details refer to the KDM specification [3].
- Project-system: The user can open a new or existing project, which saves the added folders and files in a list. After program start a new unnamed project is open.
- Folders and files with system artifacts can be added or removed from an internal list. Items in the list are used for the architecture reconstruction.
- While the system is busy with the reconstruction, a progress-bar is shown with information about the progress and in which phase of reconstruction the system is currently working.
- After the reconstruction has finished the reconstructed architecture is opened with an XML Editor.
- Extension mechanism that lays the foundations for applying different programming languages and further abstraction mechanisms.

1.2 Desired features

- Different language support (English and German)

1.3 Optional features

- Graphical representation of the reconstructed architecture
- A scripting language which enables configuration of the items mapped to KDM (for instance, filtering of components with special names)
- Support for different functional programming languages as input (namely COBOL, FORTRAN and ANSI C)
- Support for different object-oriented programming languages as input (namely C++)
- Adding more supported programming languages as input by writing own parser for Source Navigator NG.

2 Scope

2.1 Application range

KADis can be used for reconstruction of a software architecture from source code. It is aimed that different object-oriented languages will be supported.

2.2 Target group

The target group mostly consists of software engineers and software reengineers but every person with knowledge about programming can use this tool.

3 Minimum Requirements

3.1 Software

- Java Runtime Environment (at least version 1.6 update 18)
- Any operating system that supports Java
- Eclipse (version 3.5)

3.2 Hardware

- Personal computer with minimum requirements:
  - CPU: at least Pentium 4 with 2 GHz
  - RAM: at least 2 GB
  - Graphic-card: at least DirectX 9 compatible
  - Free HDD space for the resulting KDM-file (min. 500 MB)

4 Product Overview

Figure 1 gives an overview of User Use Cases and System Use Cases. The User Use Cases define the possible interaction between the user and KADis. Section 6.1 describes them in detail. The System Use Cases define the interaction between KADis and external services. For a specific description refer to Section 6.2. All displayed Use Cases are required features.

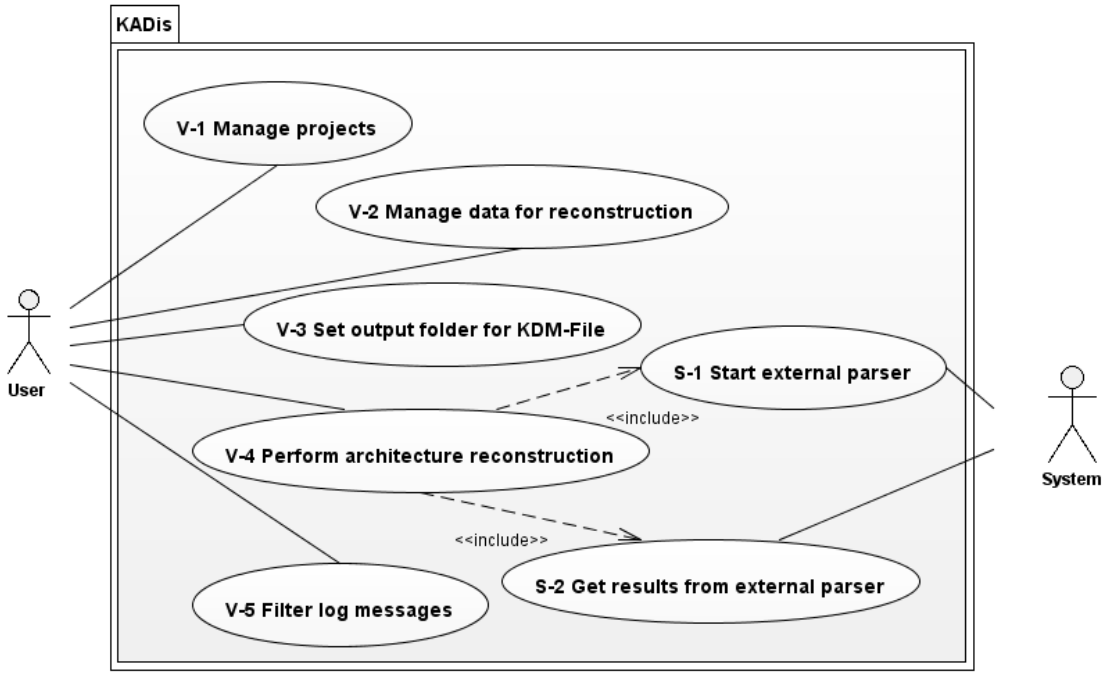

Figure 1: Overview of User Use Cases and System Use Cases

5 Actors

The following actors interact with KADis. They relate to the Use Cases described in Section 6.

Actor	Description	Related Use Cases
User	Manages projects, folders and files. Starts the architecture reconstruction process.	• V-1 Manage projects • V-2 Manage data for reconstruction • V-3 Set output folder for KDM-file • V-4 Perform architecture reconstruction • V-5 Filter log messages
System	Works with the external parser and conducts the architecture reconstruction process.	• S-1 Parse artifacts • S-2 Get results from external parser

6 Product Functions

6.1 User Use Cases

The Use Cases listed below are described in the following:

- V-1 Manage projects
- V-2 Manage data for reconstruction
- V-3 Set output folder for KDM-file
- V-4 Perform architecture reconstruction
- V-5 Filter log messages

6.1.1 Manage projects

Use Case number	V-1
Use Case name	Manage projects
Primary actor	User
Other actors	-
Description	The user creates a new project or, opens or saves an existing project.
Precondition	-
Postcondition	The created or chosen project is open.
Functionality of Use Case	Steps: 1. The user chooses ”New project“ from the file menu. 2. The user enters a name for the new project.
Alternatives	to 1) The user chooses ”Open project“ from the file menu. to 2) The user chooses ”Save project“ from the file menu.
Exception	The user did not enter a valid project name.
Used Use Cases	-

6.1.2 Manage data for reconstruction

Use Case number	V-2
Use Case name	Manage data for reconstruction
Primary actor	User
Other actors	-
Description	The user manages the data used for reconstruction.
Precondition	-
Postcondition	The selected data was added to or removed from the data list.
Functionality of Use Case	Steps: 1. The user clicks on ”Add folder“. 2. The user chooses a folder in the opening dialogue.
Alternatives	to 1) The user clicks on ”Add file“. to 1) The user clicks on ”Remove“.
Exception	-
Used Use Cases	-

6.1.3 Set output folder for KDM-file

Use Case number	V-3
Use Case name	Set output folder for KDM-file
Primary actor	User
Other actors	-
Description	The user sets the output folder in which the created KDM-file will be saved.
Precondition	-
Postcondition	The output path is set to a valid location.
Functionality of Use Case	Steps: 1. The user clicks on ”Change“ beneath the displayed output folder. 2. The user chooses a folder in the opening dialogue.
Alternatives	-
Exception	-
Used Use Cases	-

6.1.4 Perform architecture reconstruction

Use Case number	V-4
Use Case name	Start architecture reconstruction
Primary actor	User
Other actors	System
Description	The user performs the software architecture reconstruction process and the system executes the reconstruction.
Precondition	-
Postcondition	A new KDM-file is stored in the output folder. The name of the file is the project name and a timestamp of the start time of the reconstruction process. The created KDM-file was opened in a XML Editor.
Functionality of Use Case	Steps: 1. The user clicks on ”Start reconstruction“. 2. The system starts the external parser (6.2.1) 3. The system processes the results from the external parser (6.2.2)
Alternatives	-
Exception	No data was selected for reconstruction.
Used Use Cases	S-1 and S-2 (see Section 6.2.1 and 6.2.2)

6.1.5 Filter log messages

Use Case number	V-5
Use Case name	Filter log messages
Primary actor	User
Other actors	-
Description	The user filters different types of log messages.
Precondition	The check-box with the desired filter is unchecked.
Postcondition	The log window displays only the filtered messages.
Functionality of Use Case	Steps: 1. The user clicks on the checkbox ”Errors“.
Alternatives	1. to 1) The user clicks on the checkbox ”Warnings“. 2. to 1) The user clicks on the checkbox ”Normal“.
Exception	-
Used Use Cases	-

6.2 System Use Cases

The following Use Cases are described in the subsequent Sections 6.2.1 and 6.2.2:

- S-1 Parse artifacts
- S-2 Get results from external parser

6.2.1 Parse artifacts

Use Case number	S-1
Use Case name	Parse artifacts
Primary actor	System
Other actors	-
Description	The system parses the artifacts.
Precondition	-
Postcondition	The parser ran and a result file is available.
Functionality of Use Case	Steps: 1. The system starts parsing the artifacts.
Alternatives	-
Exception	The parser was not found.
Used Use Cases	-

6.2.2 Get results from external parser

Use Case number	S-2
Use Case name	Get results from external parser
Primary actor	System
Other actors	-
Description	The results from parsing step are fetched from a result file.
Precondition	The parser ran and a result file is available.
Postcondition	The results of the parser are loaded in the system.
Functionality of Use Case	Steps: 1. The system opens the result file and reads it.
Alternatives	-
Exception	Result file could not be read.
Used Use Cases	-

7 Product Data

The following Section declares all data and data structures that KADis saves and uses:

7.1 Projects

Different projects for different reconstructions can be defined. The data saved for every project is:

- Name
- Folders and files as paths that were added
- Output folder for the created KDM-file
- The path to the last reconstructed KDM-file (if there is one)

7.2 KDM-files

The reconstructed architecture is stored in a file. For the specific data, which is stored, refer to KDM specification [3]. The output includes an inventory model, a code model, and a structure model from KDM.

8 Product Performance

Scoping: By adding own folders and files the user can determine which data will be used for architecture reconstruction. In this way the user has control over the artifacts to focus on. For instance, only one subsystem can be selected and reconstructed to focus on this special subsystem.

Abstraction: In a system with multi million lines of source code it is not adequate to view every single class. Instead of this, the classes must be clustered to components, for instance. The different views of the architecture must be reconstructed. Abstraction mechanisms make this possible. The software will make basic abstraction (namely by simple clustering) of the system artifacts. Other abstraction mechanisms can be implemented through an interface.

9 Graphical User Interface

This Section provides an overview and first impression of the Graphical User Interface of KADis. The sketched Graphical User Interface is only a prototype for showing the functionality and constitutes a coarse overview of the final user interface.

9.1 Main window

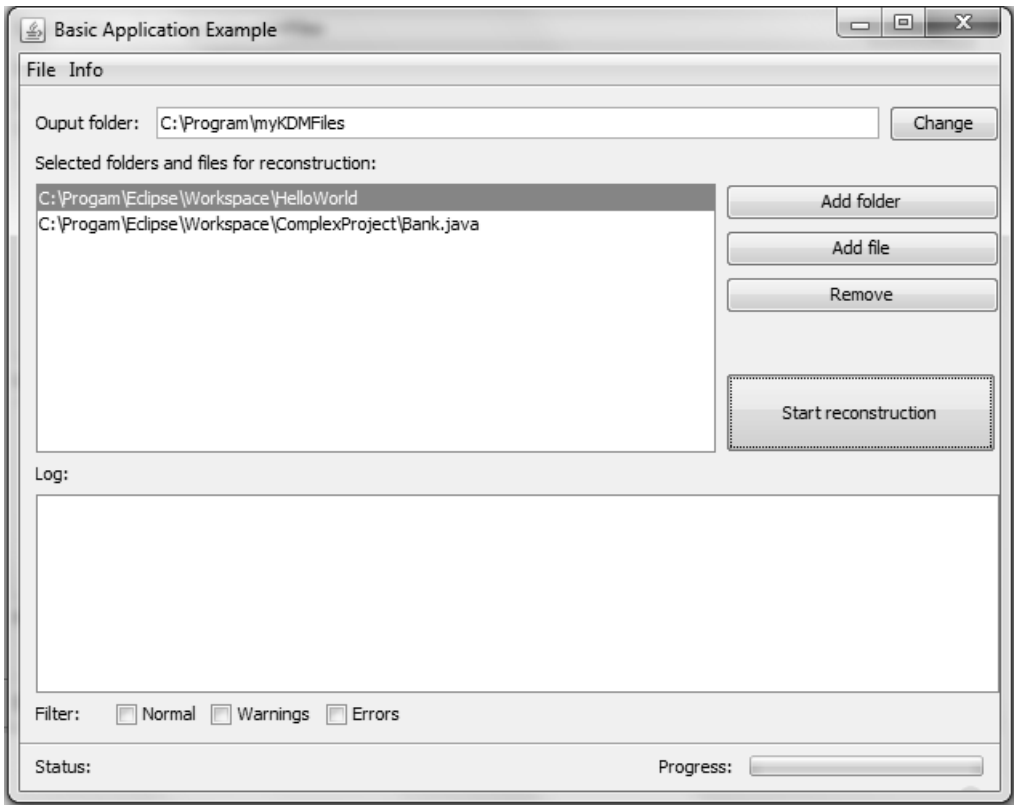

Figure 2: Main window

Figure 2 shows the main window. The button ”Change“ changes the output folder (Use Case: V-3 Set output folder for KDM-file). The buttons ”Add folder“, ”Add files“ and ”Remove“ realize the adding and removing of folders and files used for reconstruction (Use Case: V-2 Manage data for reconstruction). The ”Start reconstruction“ button starts the reconstruction process (Use Case: V-4 Perform architecture reconstruction). The log window displays normal text in black, warnings in a dark blue and errors in a dark red color. The filter options under the log window filter the log messages (Use Case: V-5 Filter log messages). Status at the bottom shows statuses like ”Reconstruction started“ or in which phase the reconstruction process currently is. The progress-bar at the lower right shows the overall progress of reconstruction process.

9.2 Dialogues

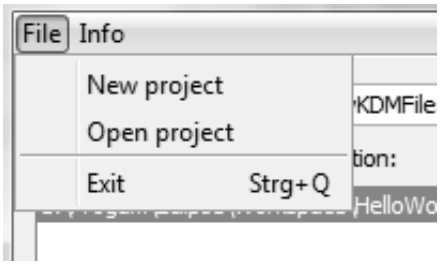

Figure 3: File menu

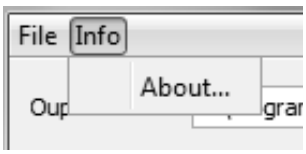

Figure 4: Info menu

1. The file menu contains the items shown in Figure 3. The first two items, namely ”New project“ and ”Open project“, realize the project-system as described in the Use Case V-1 Manage projects. The ”Exit“ item closes the application.

2. The info menu consists of the items shown in Figure 4. The ”About“ item opens a new window with details about KADis (e.g., version and author).

9.3 Layout

To enable an intuitive user interaction with the program, the layout will be guided by programming-guidelines for designing an user interface for desktop applications like [2].

9.4 Eclipse integration

Eclipse integration will be made to enable users a familiar handling with the program. The program will be implemented as an Eclipse-RCP application. The detailed integration is still under examination.

10 Software System's Attributes

	very important	important	less important	unimportant
robustness	x			
reliability	x			
maintainability	x			
extensibility	x			
user friendliness		x		
efficiency		x		
portability		x		
compatibility		x		

Table 1: Software system's attributes

11 Test Cases

The following scenarios will be tested to verify system functionality:

11.1 Hello World program

A simple "Hello World program" will be created in Java with two classes and an association between them. They will be located in a package called *helloWorldPackage*. The information has to be present in the created KDM-file.

11.2 More complex test program

A more complex test program will be created in Java. This program will consist of two classes in one package called *firstPackage* and two classes in another package called *secondPackage*. The classes will call each other. The information must be contained in the created KDM-file either.

11.3 JPetStore

JPetStore [1] will be used to test the program. All of the static information in JPetStore must be contained in the created KDM-file.

12 Development Environment

12.1 Software

- Platform:
  - Java
  - Subversion
- Tools:
  - Eclipse
  - Adobe Acrobat Reader
  - LaTeX
  - Visual Paradigm
  - NetBeans with UML plugin
- Operating system:
  - Windows 7

12.2 Hardware

- One personal computer that meets the minimum requirements listed in Section 3 with Internet access.

References

[1] iBATIS. JPetstore. `http://apache.linux-mirror.org/ibatis/binaries/ibatis.java/JPetStore-5.0.zip`, 2010-02-15.

[2] Microsoft. Microsoft Inductive User Interface Guidelines. `http://msdn.microsoft.com/en-us/library/ms997506.aspx`, 2010-02-15.

[3] Object Management Group. Knowledge Discovery Meta-Model. `http://www.omg.org/spec/KDM/1.1/`, 2010-02-15.

[4] sourcenav NG development group. Source Navigator NG. `http://sourcenav.berlios.de/`, 2010-02-15.

Appendices

A Acronyms

KADis
: KDM Architecture Discoverer

KDM
: Knowledge Discovery Meta-Model

B Glossary

Desired feature

The implementation of this feature is not critical but should be implemented.

Optional feature

The feature will be implemented only if there is enough time.

Required feature

A feature that is critical and has to be implemented.

System artifact

System artifacts are things like source code, build files, configuration files etc.

System Use Cases

Use Cases that have the system as their primary actor.

User Use Cases

Use Cases that have the user as their primary actor.

E Design Specification

Design Specification:

KDM Architecture Discoverer (KADis)

State: 2010-02-15
Version: 1.0

Author:

Florian Fittkau

Contents

1 Introduction **1**
1.1 References 1
1.2 Overview 1

2 System Overview **2**
2.1 Packages 2
2.2 Components 3

3 Packages **4**
3.1 View 4
3.1.1 EclipseRCP 4
3.1.1.1 MenuHandler 4
3.1.1.2 Preferences 4
3.1.1.3 XMLEditor 4
3.2 Model 4
3.2.1 ObserverInterfaces 4
3.2.2 Repository 4
3.3 Controller 4
3.3.1 Abstraction 5
3.3.2 Cache 5
3.3.2.1 Cache Model 5
3.3.2.2 Cache Types 5
3.3.3 Parser 5
3.3.3.1 Parser Helper 5
3.3.3.2 SourceNav 5
3.3.3.3 SourceNav Handler 5
3.3.3.4 SourceNav Models 5
3.3.4 SARManager 5
3.3.5 Tools 6

4 Classes **7**
4.1 View (Section B.1) 7
4.1.1 EclipseRCP (Section B.1.1) 7
4.1.1.1 MenuHandler (Section B.1.1.1) 8
4.1.1.2 Preferences (Section B.1.1.2) 8
4.1.1.3 XMLEditor (Section B.1.1.3) 8
4.2 Model (Section B.2) 9

4.2.1 ObserverInterfaces (Section B.2.1) . . . 10
4.2.2 Repository (Section B.2.2) . . . 10
4.3 Controller (Section B.3) . . . 10
4.3.1 Abstraction (Figure B.3.1) . . . 10
4.3.2 Cache (Section B.3.2) . . . 10
4.3.2.1 CacheModel (Section B.3.2.1) . . . 11
4.3.2.2 CacheTypes (Section B.3.2.2) . . . 11
4.3.3 Parser (Section B.3.3) . . . 11
4.3.3.1 Parser Helper (Section B.3.3.1) . . . 12
4.3.3.2 SourceNav (Section B.3.3.2) . . . 12
4.3.3.3 SourceNav Handler (Section B.3.3.3) . . . 12
4.3.3.4 SourceNav Models (Section B.3.3.4) . . . 13
4.3.4 SARManager (Section B.3.4) . . . 13
4.3.5 Tools (Section B.3.5) . . . 13

5 Dynamic Diagrams 14
5.1 V-1 Manage projects . . . 14
5.2 V-2 Manage data for reconstruction . . . 15
5.3 V-3 Set output folder for KDM-file . . . 16
5.4 V-4 Perform architecture reconstruction . . . 17
5.5 V-5 Filter log messages . . . 18
5.6 S-1 Parse artifacts . . . 19
5.7 S-2 Get results from external parser . . . 20

References 21

A Acronyms i

B Class Diagrams ii
B.1 View class diagram . . . ii
B.1.1 EclipseRCP class diagram . . . iii
B.1.1.1 MenuHandler class diagram . . . iv
B.1.1.2 Preferences class diagram . . . v
B.1.1.3 XMLEditor class diagram . . . vi
B.2 Model class diagram . . . vii
B.2.1 ObserverInterfaces class diagram . . . viii
B.2.2 Repository class diagram . . . ix
B.3 Controller class diagram . . . x
B.3.1 Abstraction class diagram . . . xi
B.3.2 Cache class diagram . . . xii

B.3.2.1 CacheModel class diagram xiv
B.3.2.2 CacheTypes class diagram xv
B.3.3 Parser class diagram xvi
B.3.3.1 Parser Helper class diagram xvii
B.3.3.2 SourceNav class diagram xviii
B.3.3.3 SourceNav Handler class diagram xix
B.3.3.4 SourceNav Models class diagram xx
B.3.4 SAR Manager class diagram xxi
B.3.5 Tools class diagram xxii

1 Introduction

This document represents a documentation of the design of KDM Architecture Discoverer (KADis). It is written particularly for people who want to develop KADis further. For this purpose the understanding of the underlying design is essential. There already exists a functional specification [3], which describes the fundamental functions of the application. Furthermore, the different Use Cases are defined in the functional specification, which give a more precisely overview of the planed program features.

1.1 References

To enable an easy development and an open project format Eclipse is used as IDE. Eclipse [1] is a free platform, which has native Java support. Furthermore, the application is designed and implemented as an Eclipse RCP application.

KADis is developed with Java 1.6. Thus, it is assumed that the reader is familiar with Java. Java is a free programming language which was developed by Sun Microsystems. More information about Java and Sun Microsystems can be obtained from the official website [7].

For modeling the system the Unified Modelling Language (UML) plug-in for NetBeans is used. This plug-in for the free IDE NetBeans is specialized for the creation of UML diagrams. NetBeans plug-ins and further information can be found on the NetBeans website [5].

The design utilizes some design patterns that are described by Gamma et al [4].

For data collection the tool Source Navigator NG is used. More information about this tool is available on its website [8].

To compensate the shortcomings of Source Navigator NG KADis features a Java AST implementation from the Eclipse project [2].

1.2 Overview

Section 2 gives a short, abstract overview of the structure of the system. In Section 3 the description of the packages follows. Section 4 displays the classes and Section 5 describes the dynamic behavior of the system on the basis of its Use Cases.

2 System Overview

2.1 Packages

KADis implements the MVC pattern. Therefore, KADis consists of three packages, namely a view, a model and a controller package (Figure 1). In the view package all graphical components are contained. The controller package covers the logical aspects and tasks. All data of KADis is held in the model package and when this data has changed, the view is informed by it.

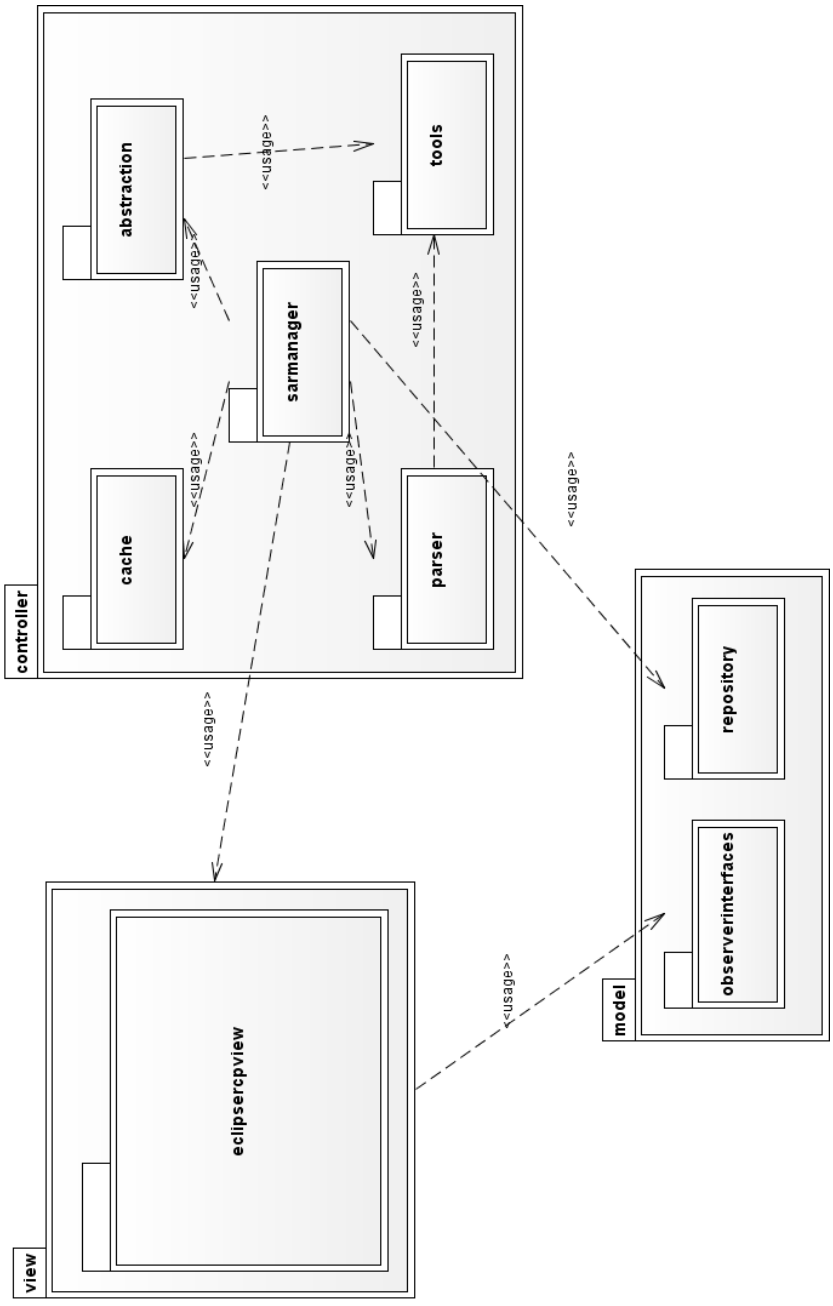

Figure 1: Packages of KADis

2.2 Components

Figure 2 sketches the components of KADis. The MVC components described in Section 2.1 can be found here again.

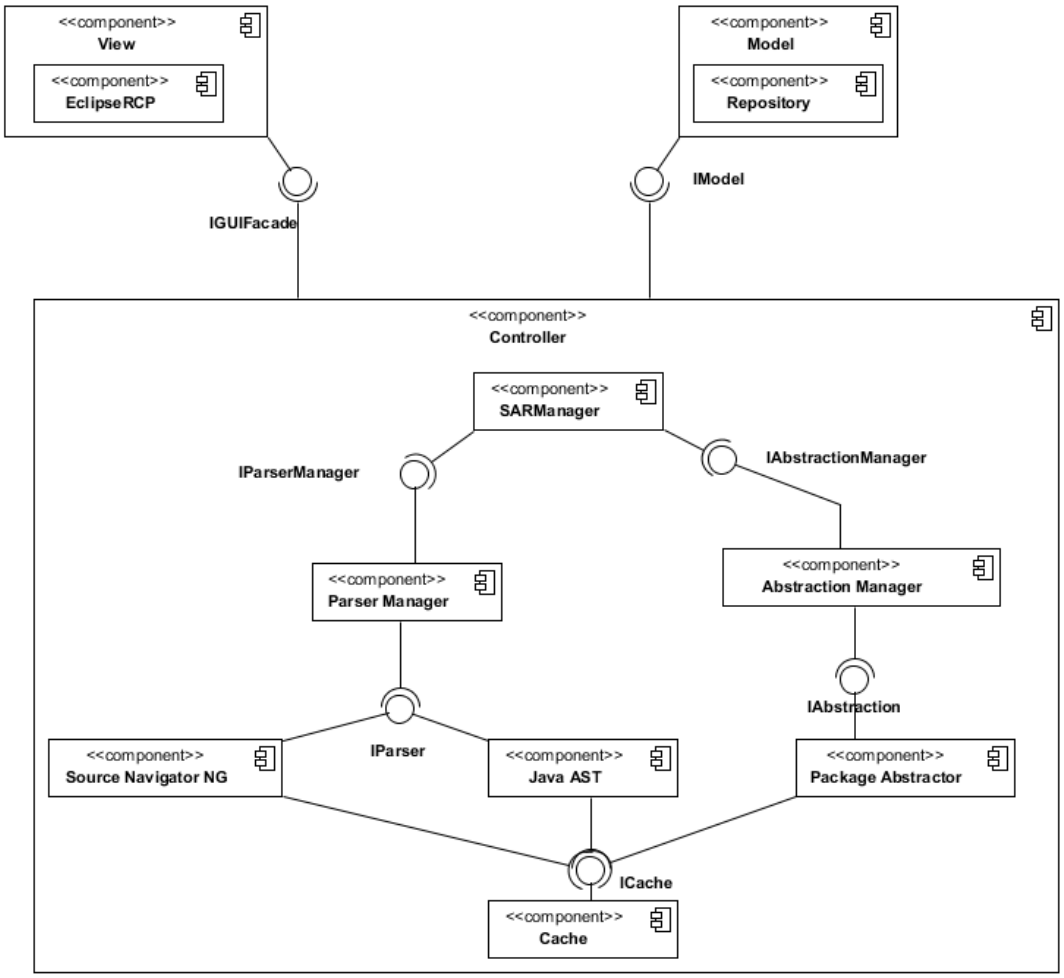

Figure 2: Components of KADis

3 Packages

This Section describes the different packages of KADis.

3.1 View

The view package realizes the graphical user interface.

3.1.1 EclipseRCP

This package realizes the displaying of a user interface in an Eclipse RCP application.

3.1.1.1 MenuHandler

All menu handlers are contained in this package. A menu handler is called by a click on an item in the menu bar of KADis.

3.1.1.2 Preferences

This package holds the preferences dialog and constants for it.

3.1.1.3 XMLEditor

In this package the XML editor is contained. The XML editor is created with a template example of Eclipse RCP applications.

3.2 Model

The model holds all data in KADis.

3.2.1 ObserverInterfaces

The observer interfaces for the view are contained in this package.

3.2.2 Repository

The repository package contains the different representations of the final output of the Software Architecture Reconstruction (SAR) process.

3.3 Controller

The controller handles all requests from the view.

3.3.1 Abstraction

This package contains the elements involved in the abstraction phase of the SAR process.

3.3.2 Cache

The CacheOOPFacade is hold in this package. It realizes a common database for all SAR phases.

3.3.2.1 Cache Model

The cache model package contains the models generated from the different tables in the common database.

3.3.2.2 Cache Types

The cache type package contains the types that are shared by the different models in the common database.

3.3.3 Parser

The parser package holds all parsers that are used in KADis. These are at this time Source Navigator NG and Java AST.

3.3.3.1 Parser Helper

This package contains helper classes for common parser jobs.

3.3.3.2 SourceNav

The sourcenav package holds all classes that get and convert results from Source Navigator NG.

3.3.3.3 SourceNav Handler

The different table handlers for Source Navigator NG are contained here.

3.3.3.4 SourceNav Models

The different models for Source Navigator NG are held by this package.

3.3.4 SARManager

The SAR manager handles the correct sequence of the different phases in the SAR process.

3.3.5 Tools

Different tools, for instance for accessing the local file system, are contained in this package.

4 Classes

This Section lists the classes of KADis. The class diagrams of each package can be found in appendix B.

4.1 View (Section B.1)

- **EclipseRCPGUIFacade:** Realizes the IGUIFacade interface to enable an Eclipse RCP Application GUI.
- **IGUIFacade:** Defines the required methods of the GUI for KADis.

4.1.1 EclipseRCP (Section B.1.1)

- **Activator:** The activator class controls the plug-in life cycle and is automatically created for an Eclipse RCP application.
- **ApplicationActionBarAdvisor:** An action bar advisor is responsible for creating, adding, and disposing of the actions and is automatically created for an Eclipse RCP application.
- **ApplicationWorkbenchAdvisor:** The workbench advisor handles settings for the created workbench and is automatically created for an Eclipse RCP application.
- **ApplicationWorkbenchWindowAdvisor:** The workbench window advisor handles settings for the main window in the workbench and is automatically created for an Eclipse RCP application.
- **ChangeOutputPanel:** This panel is for displaying the change output related components.
- **LogPanel:** The log panel shows the log and the check boxes used for filtering.
- **MainView:** The main view brings all other visible components together and creates the menu bar.
- **ManageDataPanel:** The manage data panel handles the displayed data used for SAR and has a button for starting the SAR process.
- **Perspective:** The default perspective is created because KADis has only one.
- **StatusBar:** The bottom bar that shows the current status and the overall progress.

4.1.1.1 MenuHandler (Section B.1.1.1)

- **AboutHandler:** The about handler opens the about dialog.
- **DeleteProjectHandler:** The delete project handler opens the delete project dialog.
- **ExitHandler:** This handler closes the application.
- **LoadProjectHandler:** Creates an open dialog and loads the selected dialog.
- **NewProjectHandler:** The new project handler creates a new project wizard.
- **NewProjectWizard:** The wizard that enables the creation of a new project.
- **NewProjectWizardPage:** The concrete page of the new project wizard.
- **SaveProjectAsHandler:** Opens a save as dialog.
- **SaveProjectHandler:** Saves the open project at its current location.
- **XMLEditorHandler:** Opens the XML editor.

4.1.1.2 Preferences (Section B.1.1.2)

- **kadisPreferencePage:** The concrete preference page of KADis.
- **PreferenceConstants:** Constants used for identifying the selected items.
- **PreferenceInitializer:** Creates the preference page.

4.1.1.3 XMLEditor (Section B.1.1.3)

- **ColorManager:** Automatically created from an Eclipse template for displaying an XML editor.
- **IXMLColorConstants:** Automatically created from an Eclipse template for displaying an XML editor.
- **MyXMLEditorInput:** The input provider for the XML editor is defined here.
- **NonRuleBasedDamagerRepairer:** Automatically created from an Eclipse template for displaying an XML editor.
- **TagRule:** Automatically created from an Eclipse template for displaying an XML editor.

- **XMLConfiguration:** Automatically created from an Eclipse template for displaying an XML editor.

- **XMLDocumentProvider:** Automatically created from an Eclipse template for displaying an XML editor.

- **XMLDoubleClickStrategy:** Automatically created from an Eclipse template for displaying an XML editor.

- **XMLEditor:** Automatically created from an Eclipse template for displaying an XML editor.

- **XMLPartitionScanner:** Automatically created from an Eclipse template for displaying an XML editor.

- **XMLScanner:** Automatically created from an Eclipse template for displaying an XML editor.

- **XMLTagScanner:** Automatically created from an Eclipse template for displaying an XML editor.

- **XMLWhitespaceDetector:** Automatically created from an Eclipse template for displaying an XML editor.

4.2 Model (Section B.2)

- **EGUILanguage:** Enumeration for the different languages of the GUI.

- **ELogMessageType:** The different types of log messages are defined in this enumeration.

- **LogMessage:** The log message class which enables sending log messages to the view.

- **ModelFacade:** Facade that provides all external services of the model.

- **Project:** The project class for managing data associated with a project.

- **SaveSettings:** Realizes settings that are saved and loaded when KADis shuts down or starts.

4.2.1 ObserverInterfaces (Section B.2.1)

- **IChangeOutputObserver:** The observer interface for recognizing that the output path has changed.
- **IDataHasChanged:** The observer interface for recognizing that the data used for SAR has changed.
- **ILogMessageObserver:** The observer interface for receiving log messages.
- **IProgressObserver:** The observer interface for the progress bar.
- **IProjectHasChangedObserver:** The observer interface for the title of the window which contains the project name.
- **IReconstructionEndedObserver:** The observer interface for the enabling and disabling of the start reconstruction button.
- **IStatusObserver:** The observer interface for receiving status messages.

4.2.2 Repository (Section B.2.2)

- **IRepository:** The interface for repositories
- **KDMClasses:** A repository implementation by direct mapping to KDM Classes.

4.3 Controller (Section B.3)

- **ControllerFacade:** Provides all needed methods for the view.

4.3.1 Abstraction (Figure B.3.1)

- **AbstractionManager:** Manages the abstraction process.
- **IAbstraction:** Interface for different abstraction mechanisms
- **IAbstractionManagerObs:** The observer interface for the abstraction manager.
- **PackageAbstractor:** Implements the IAbstraction interface and provides basic package abstraction.

4.3.2 Cache (Section B.3.2)

- **CacheOOPFacade:** Provides a common database for all phases of the SAR process.

4.3.2.1 CacheModel (Section B.3.2.1)

- **ClassModel:** A model for a class entity
- **FileModel:** A model for a file entity
- **ImportModel:** A model for an import entity
- **InheritanceModel:** A model for an inheritance entity
- **MethodModel:** A model for a method entity
- **PackageModel:** A model for a package entity
- **VariableModel:** A model for a variable entity

4.3.2.2 CacheTypes (Section B.3.2.2)

- **EAccess:** Enumeration for the different kinds of access to a variable (read, write,...)
- **EAttribute:** Enumeration for the modifier of classes/methods/variables (public, private,...)
- **ELanguage:** Enumeration for the concrete programming language (java, c,...)
- **EType:** Enumeration for the type of the entity if it is dynamically associated (class, method,...)
- **Position:** Position defines the start and end position of a declaration.
- **TypeHelper:** Converts from the string representation to a enumeration and vice versa.

4.3.3 Parser (Section B.3.3)

- **IParser:** Interface for different external parsers
- **IParserManagerObs:** Observer interface for the parser manager.
- **JavaAST:** Provides a Java abstract syntax tree.
- **KDMCodeModelCreator:** Creates the Knowledge Discovery Meta-Model (KDM) [6] code model.
- **KDMInventoryModelCreator:** Creates the KDM inventory model.
- **ParserManager:** Manages the parser phase.
- **SourceNavigator:** Implements the IParser interface and provides the possibility to use source navigator as external parser.

4.3.3.1 Parser Helper (Section B.3.3.1)

- **ExternalCommandExecuter:** Executes external programs.
- **IExternalProgrammFinishedObserver:** Observer interface which is called when an external program has finished
- **InputStreamGobbler:** Defines a stream thread for getting the results from an external program.
- **IResultObserver:** Observer interface for receiving the results from the input stream of an external program.

4.3.3.2 SourceNav (Section B.3.3.2)

- **SNMainTableManager:** Creates and executes the Source Navigator query script for each Source Navigator table.
- **SNResultTupleParser:** Converts the result of the query to a string array representation.

4.3.3.3 SourceNav Handler (Section B.3.3.3)

- **ClassHandler:** Handles the result of the classes table query.
- **IncludesHandler:** Handles the result of the includes table query.
- **InheritanceHandler:** Handles the result of the inheritances table query.
- **InstanceVariablesHandler:** Handles the result of the instance variables table query.
- **LocalVariablesHandler:** Handles the result of the local variables table query.
- **MethodDefintionsHandler:** Handles the result of the method definitions table query.
- **MethodImplementationsHandler:** Handles the result of the method implementations table query.
- **ProjectFilesHandler:** Handles the result of the project files table query.
- **ReferredByHandler:** Handles the result of the referred-by table query.
- **RefersToHandler:** Handles the result of the refers-to table query.
- **SymbolsOfFilesHandler:** Handles the result of the symbols-of-files table query.

4.3.3.4 SourceNav Models (Section B.3.3.4)

- **ReferredByModel:** A model that contains all referred-by relations. It should be saved into the common database in further versions.

- **RefersToModel:** A model that contains all refers-to relations. It should be saved into the common database in further versions.

4.3.4 SARManager (Section B.3.4)

- **SARManager:** Manages in which sequence the phases are executed in the SAR process.

4.3.5 Tools (Section B.3.5)

- **Filesystem:** Implements the IFilesystem interface and provides the possibility to save and load data from local hard disk.

- **IDGenerator:** A generator for unique IDs.

- **JDBCLogic:** Provides the communication to a database.

- **LanguageStrings:** The different strings for the currently selected language are fetched from this class. For instance, “Rekonstruktion” for reconstruction if German is the selected language.

- **PreferredMapper:** Enables correct mapping of XML namespaces.

- **XML:** A class for marshalling and unmarshalling XML files.

5 Realization of Use Cases: Dynamic Diagrams

The following Sections 5.1 to 5.5 describe the dynamic of the software on the basis of the Use Cases defined in the functional specification.

5.1 V-1 Manage projects

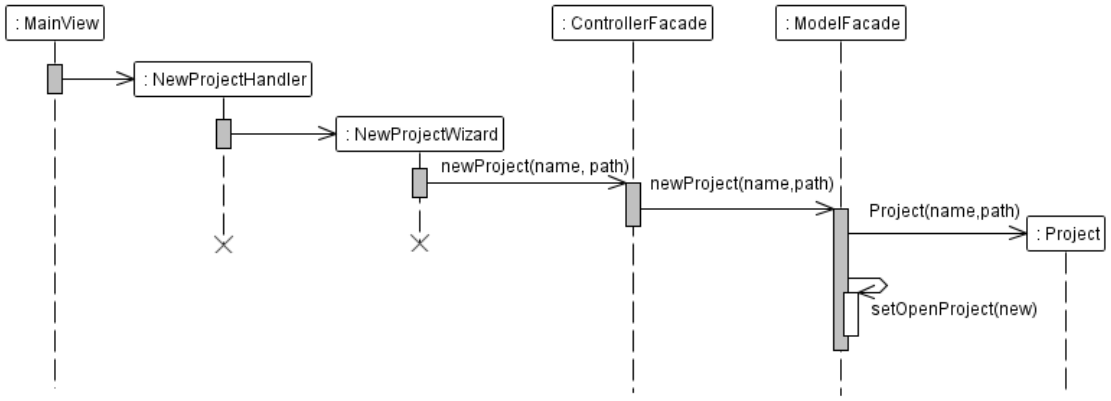

Figure 3: Create a new project

5.2 V-2 Manage data for reconstruction

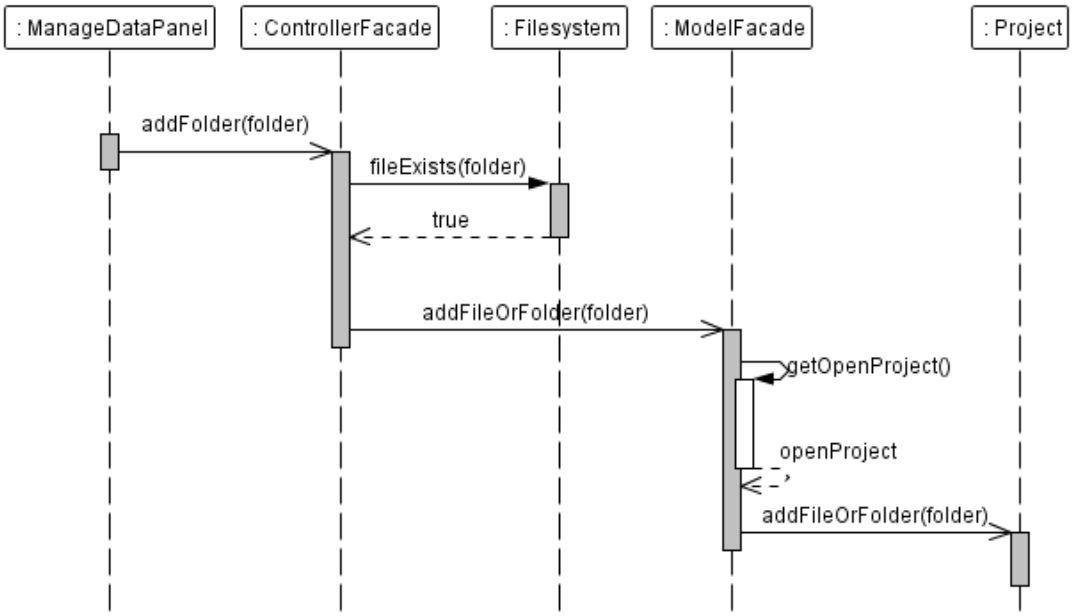

Figure 4: Add a folder to the data that is used for reconstruction

5.3 V-3 Set output folder for KDM-file

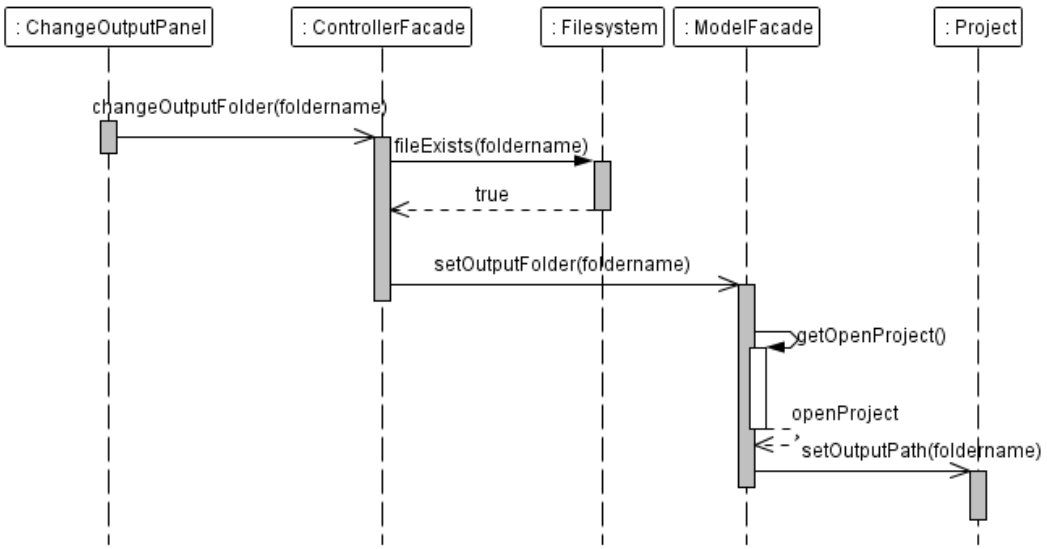

Figure 5: Change output folder

5.4 V-4 Perform architecture reconstruction

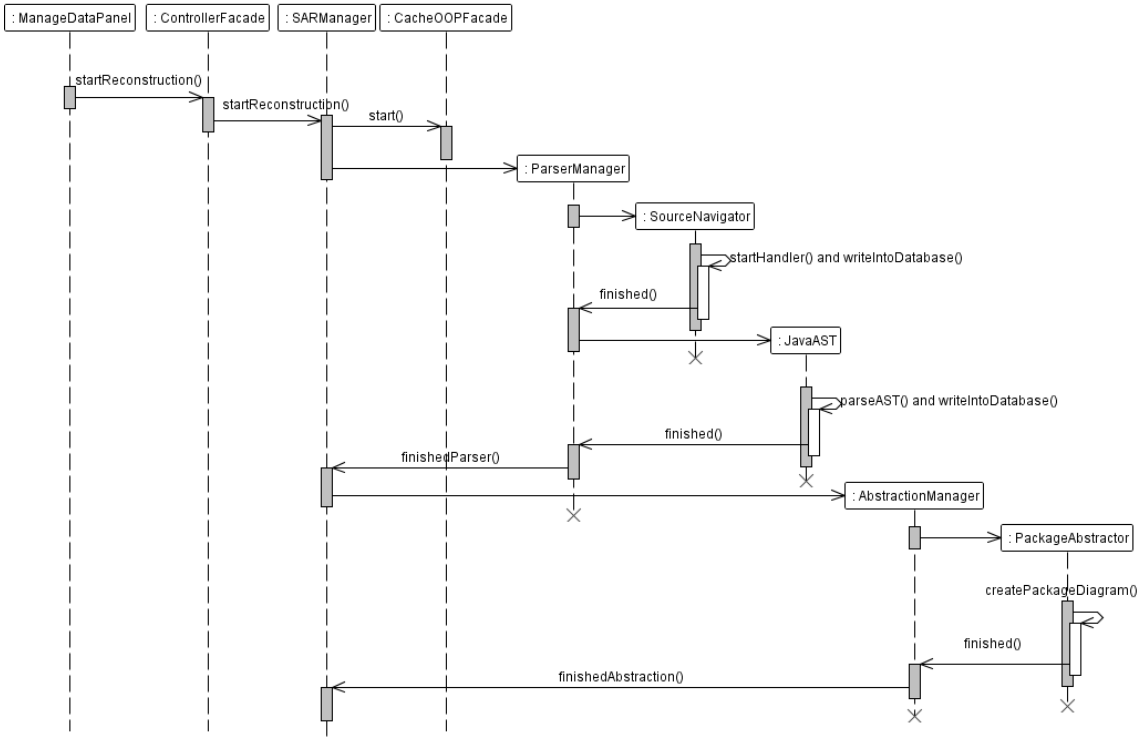

Figure 6: Perform architecture reconstruction

5.5 V-5 Filter log messages

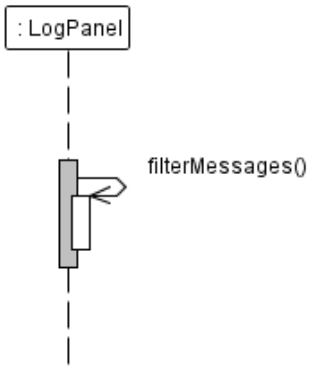

Figure 7: Filter log messages

5.6 S-1 Parse artifacts

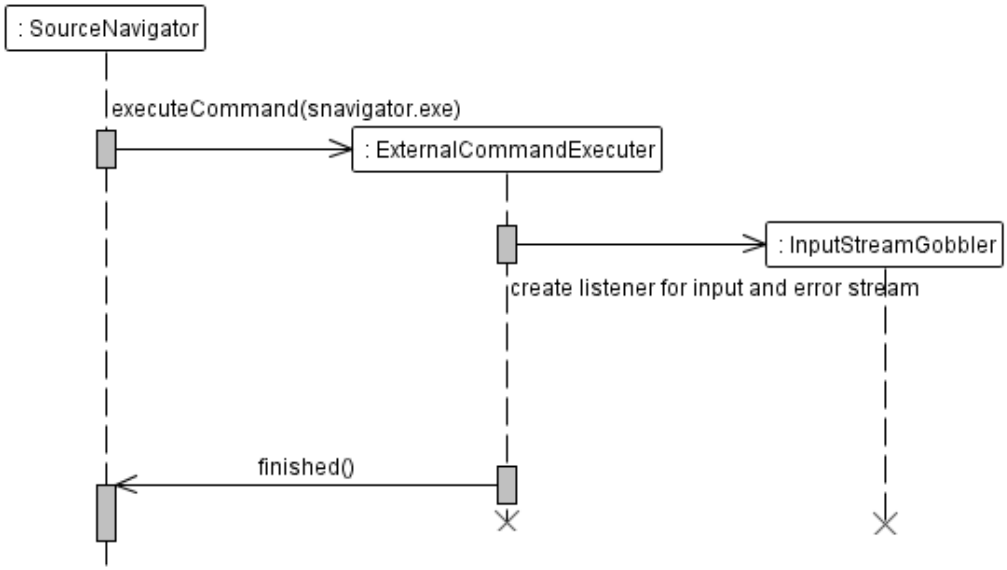

Figure 8: Start external parser

5.7 S-2 Get results from external parser

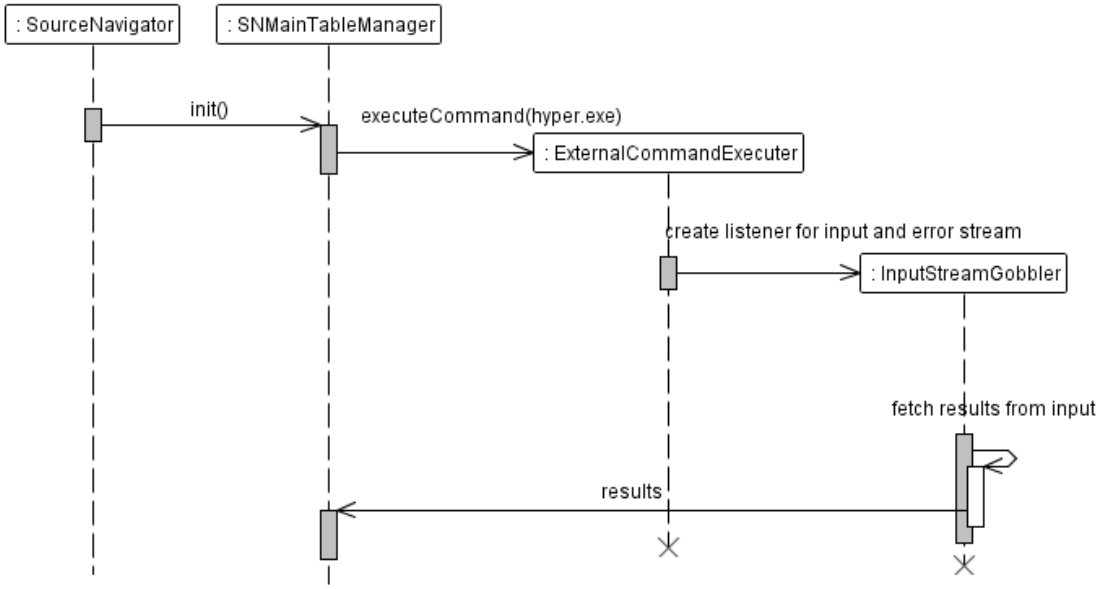

Figure 9: Get results from external parser

References

[1] Eclipse Foundation. Eclipse. `http://www.eclipse.org/`, 2010-02-15.

[2] Eclipse Foundation. Java AST. `http://help.eclipse.org/help32/nftopic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/package-summary.html`, 2010-02-15.

[3] Florian Fittkau. KADis. `http://sourceforge.net/projects/kadis/`, 2010-02-15.

[4] Gamma, Erich and Helm, Richard and Johnson, Ralph and Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, 1995.

[5] NetBeans community. NetBeans. `http://www.netbeans.org`, 2010-02-15.

[6] Object Management Group. Knowledge Discovery Meta-Model. `http://www.omg.org/spec/KDM/1.1/`, 2010-02-15.

[7] Oracle. Java. `http://www.java.com/`, 2010-02-15.

[8] sourcenav NG development group. Source Navigator NG. `http://sourcenav.berlios.de/`, 2010-02-15.

Appendices

A Acronyms

DSL
Domain-specific Language

KADis
KDM Architecture Discoverer

KDM
Knowledge Discovery Meta-Model

SAR
Software Architecture Reconstruction

UML
Unified Modelling Language

B Class Diagrams

For the specific meaning of attributes and methods refer to the JavaDoc in the source code of KADis [3].

B.1 View class diagram

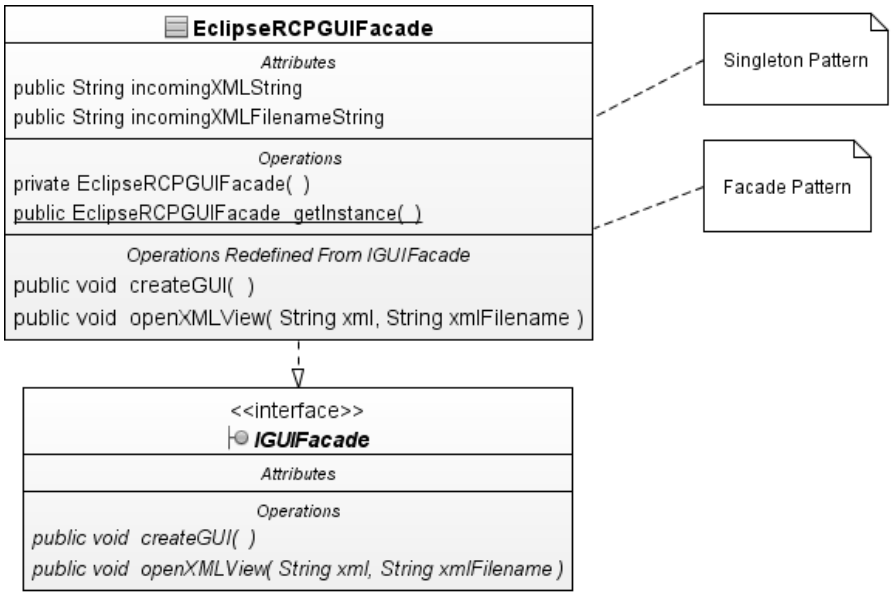

Figure 10: View class diagram

B.1.1 EclipseRCP class diagram

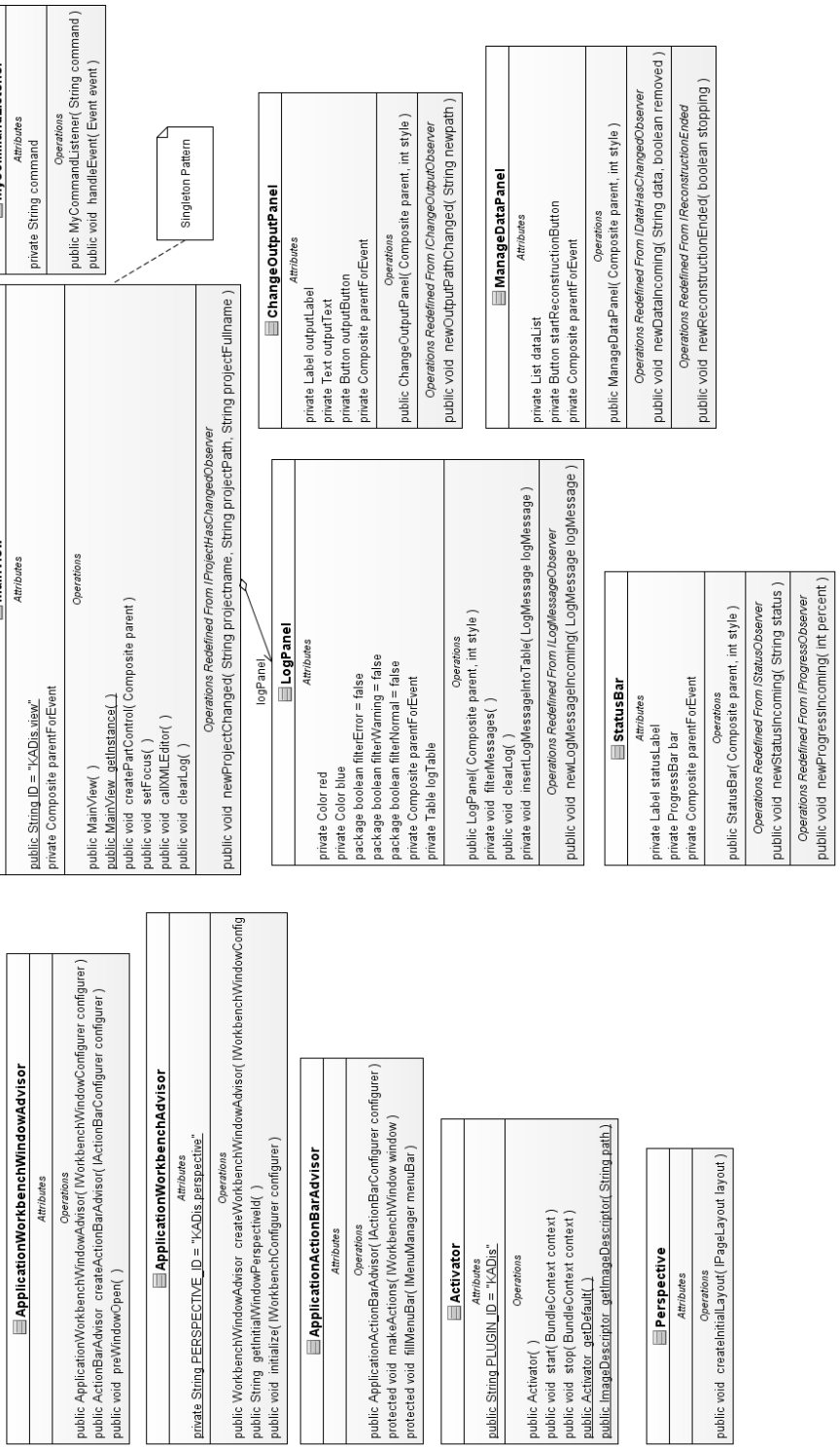

Figure 11: EclipseRCP class diagram

B.1.1.1 MenuHandler class diagram

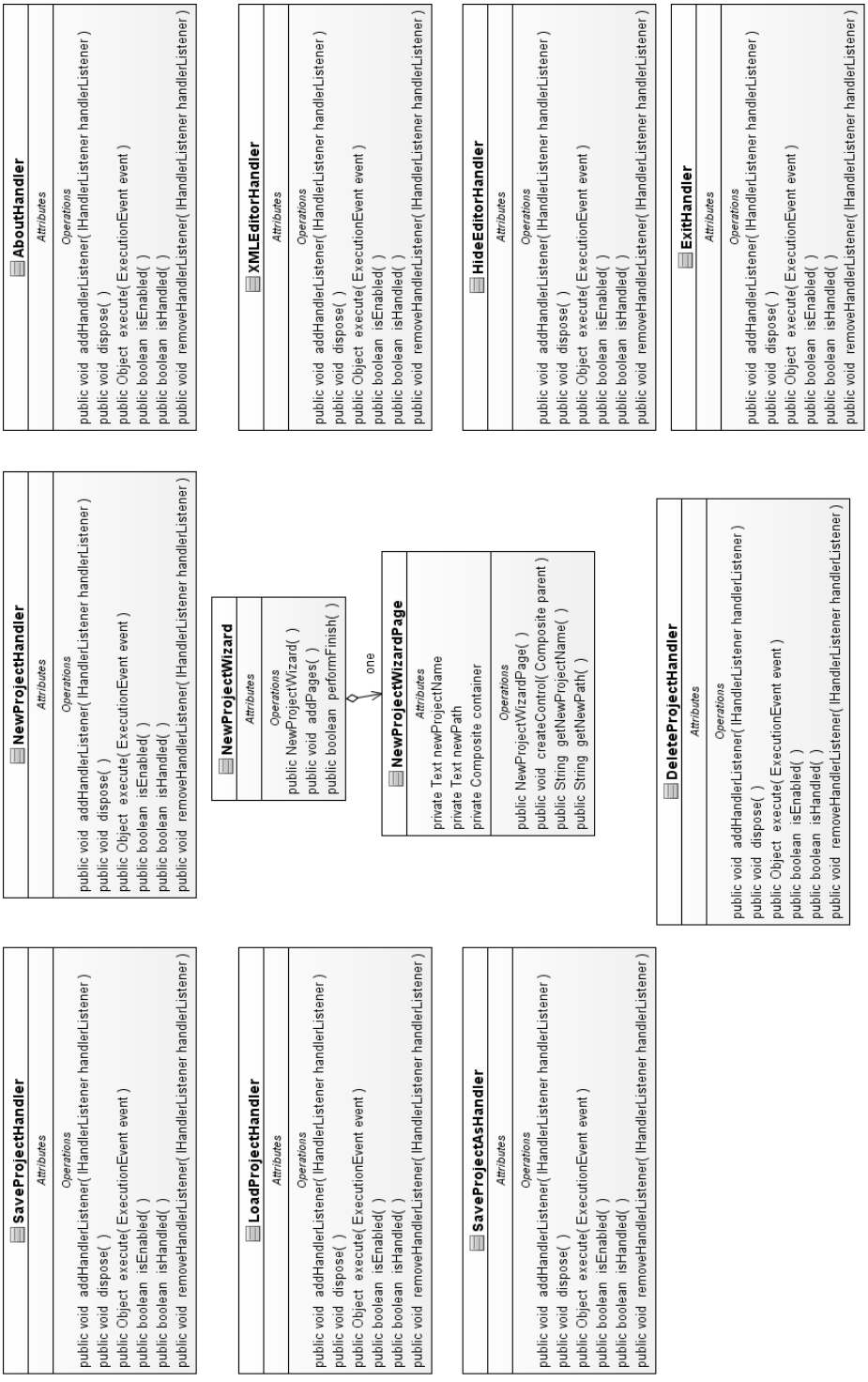

Figure 12: MenuHandler class diagram

B.1.1.2 Preferences class diagram

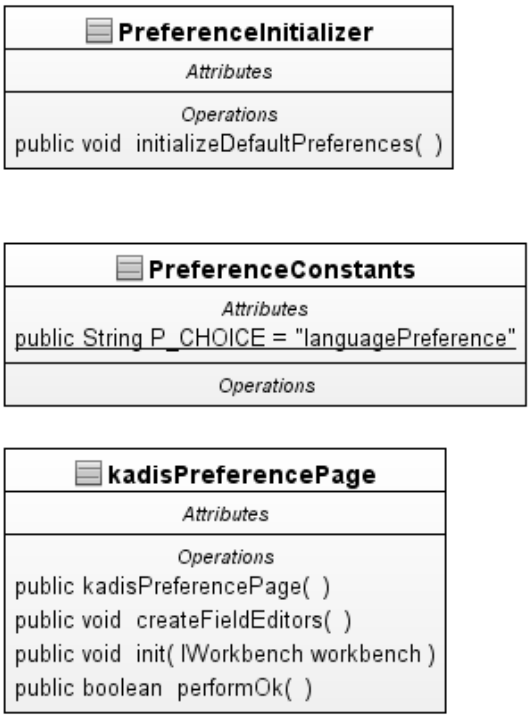

Figure 13: Preferences class diagram

B.1.1.3 XMLEditor class diagram

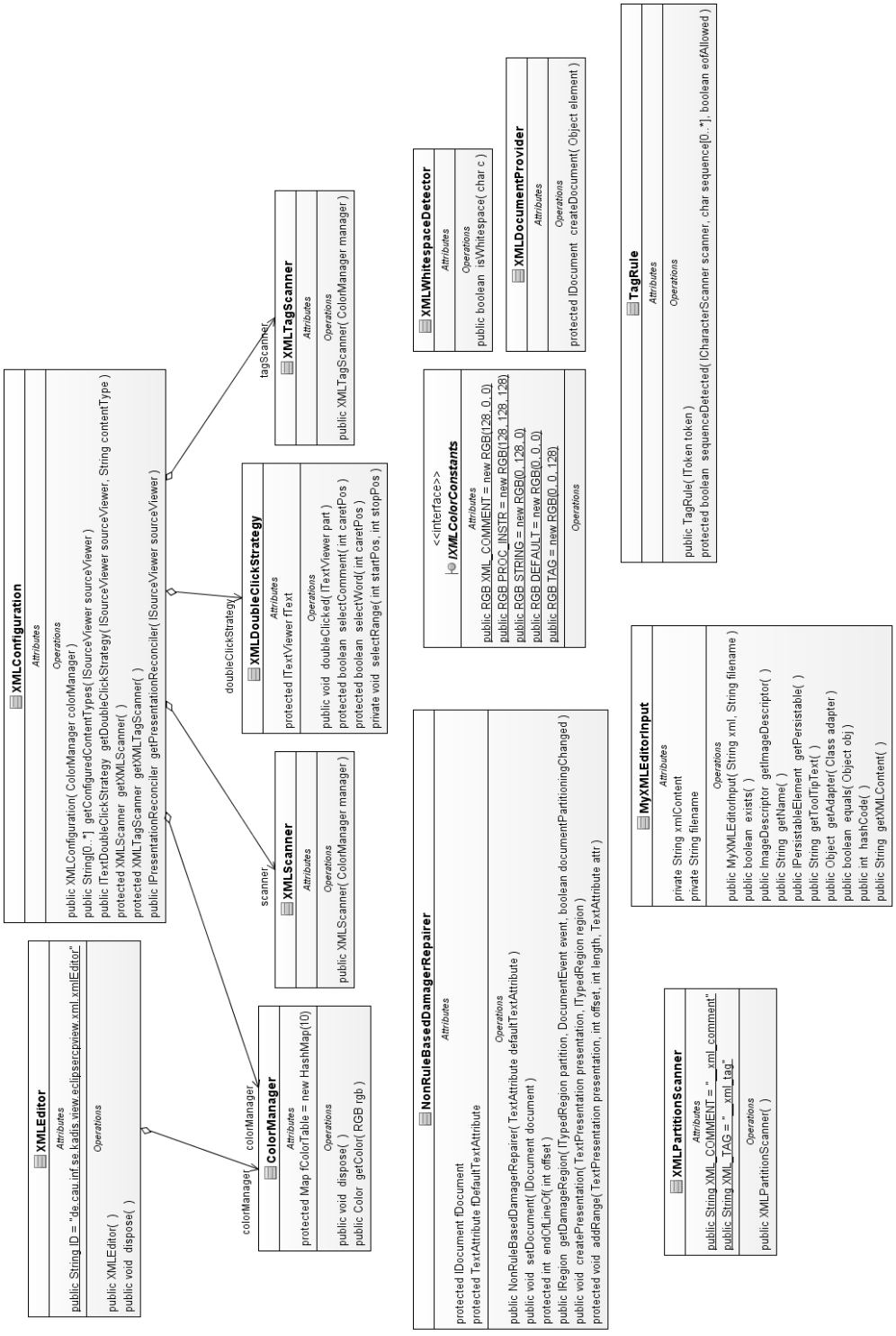

Figure 14: XMLEditor class diagram

B.2 Model class diagram

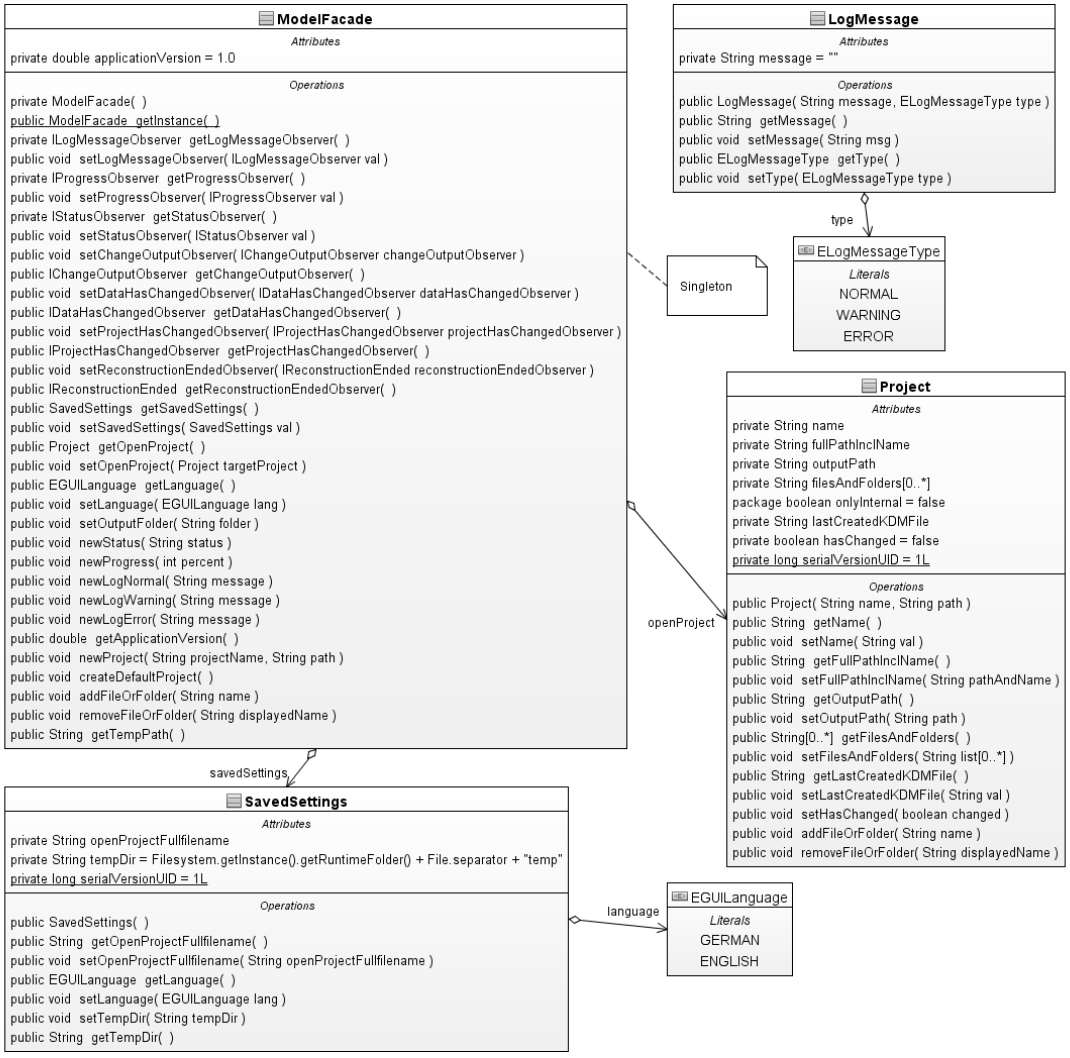

Figure 15: Model class diagram

B.2.1 ObserverInterfaces class diagram

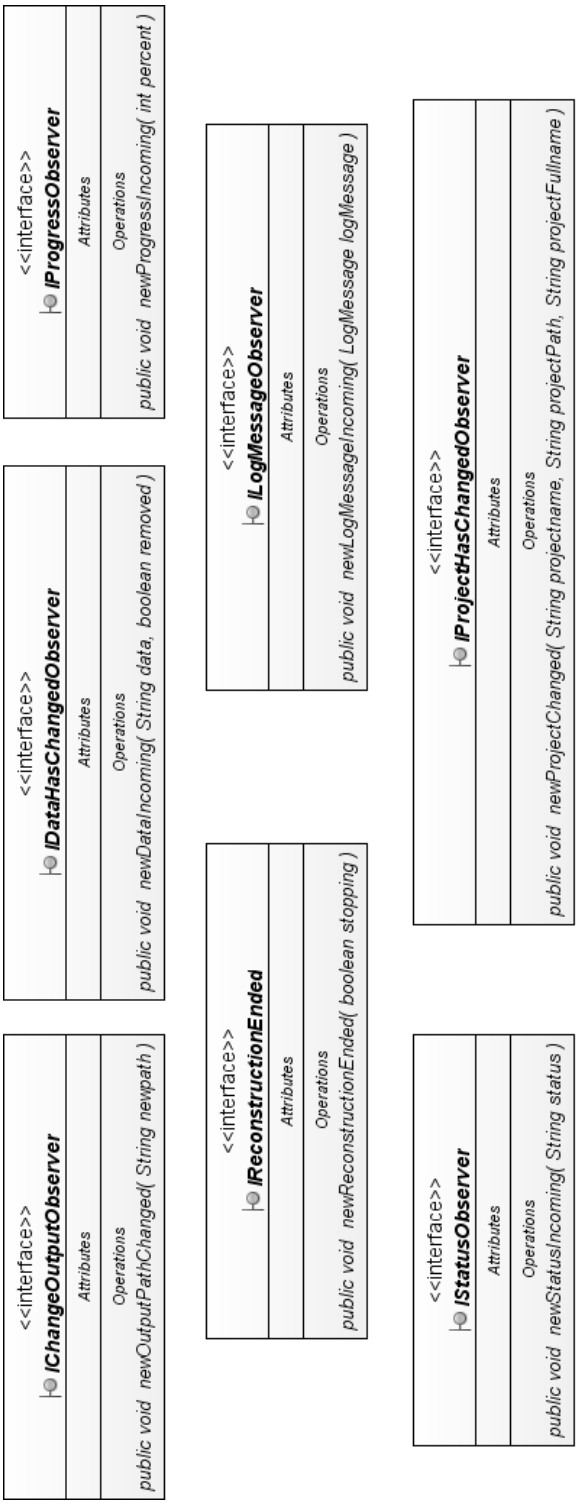

Figure 16: ObserverInterfaces class diagram

B.2.2 Repository class diagram

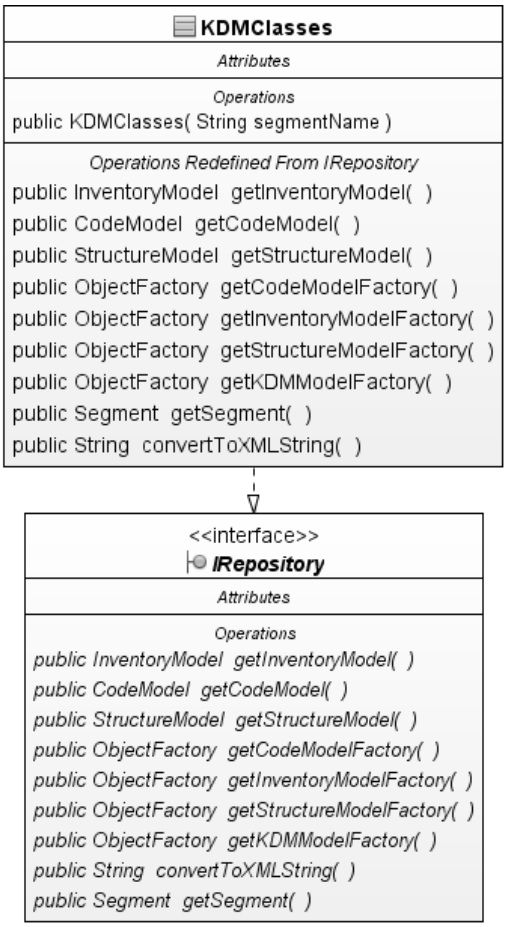

Figure 17: Repository class diagram

B.3 Controller class diagram

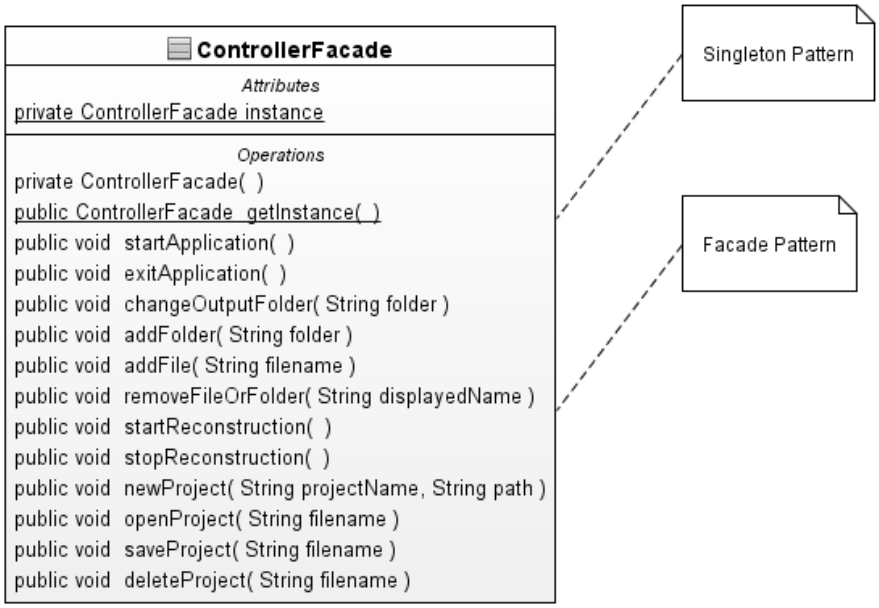

Figure 18: Controller class diagram

B.3.1 Abstraction class diagram

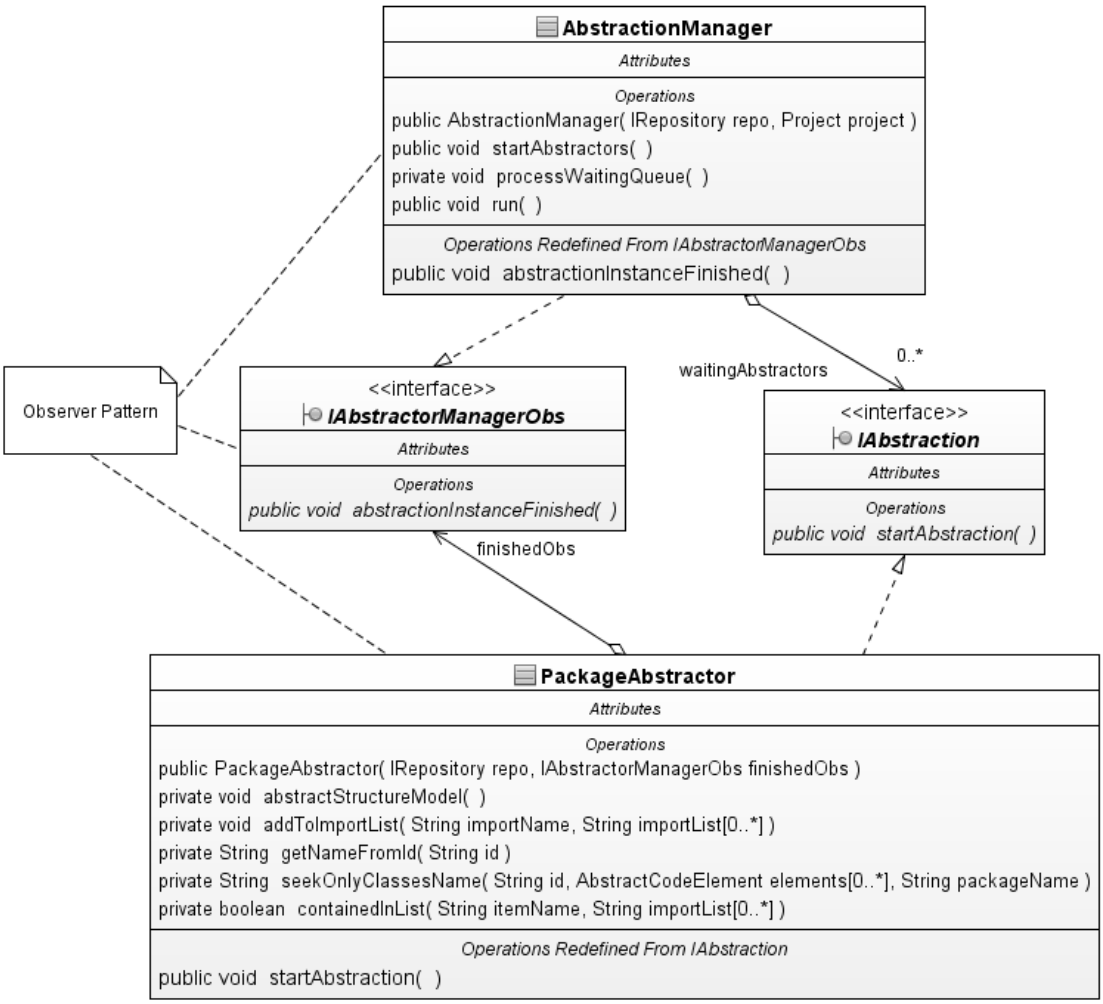

Figure 19: Abstraction class diagram

B.3.2 Cache class diagram

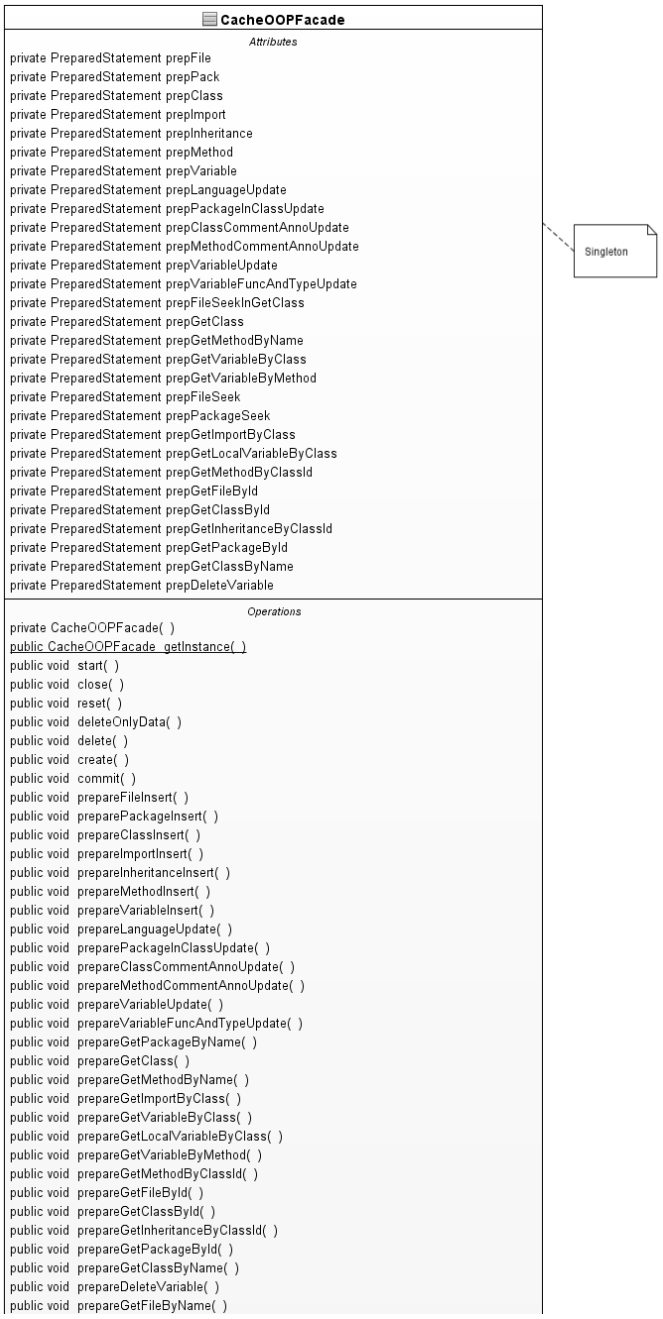

Figure 20: Cache class diagram part 1

Figure 21: Cache class diagram part 2

B.3.2.1 CacheModel class diagram

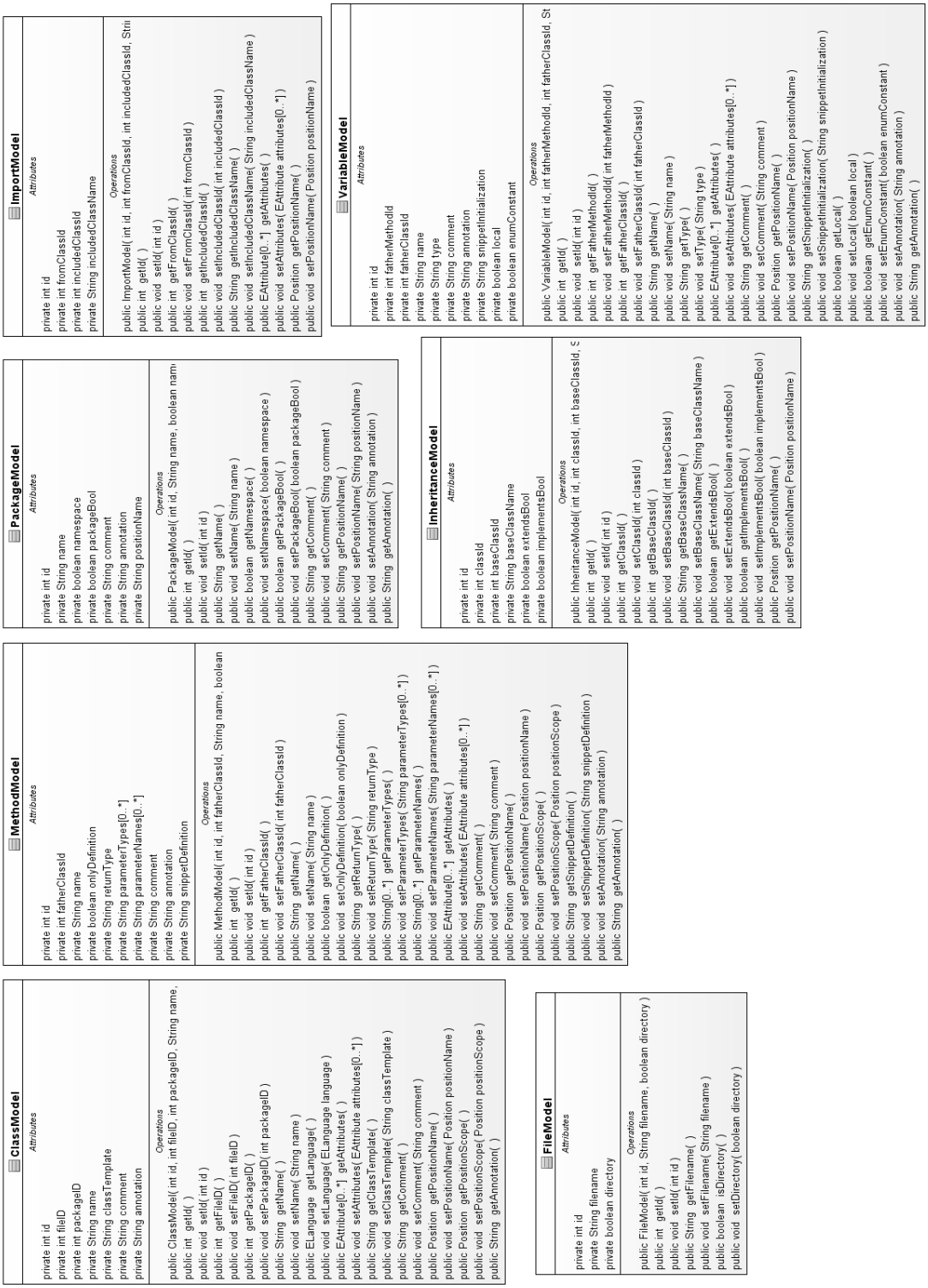

Figure 22: CacheModel class diagram

B.3.2.2 CacheTypes class diagram

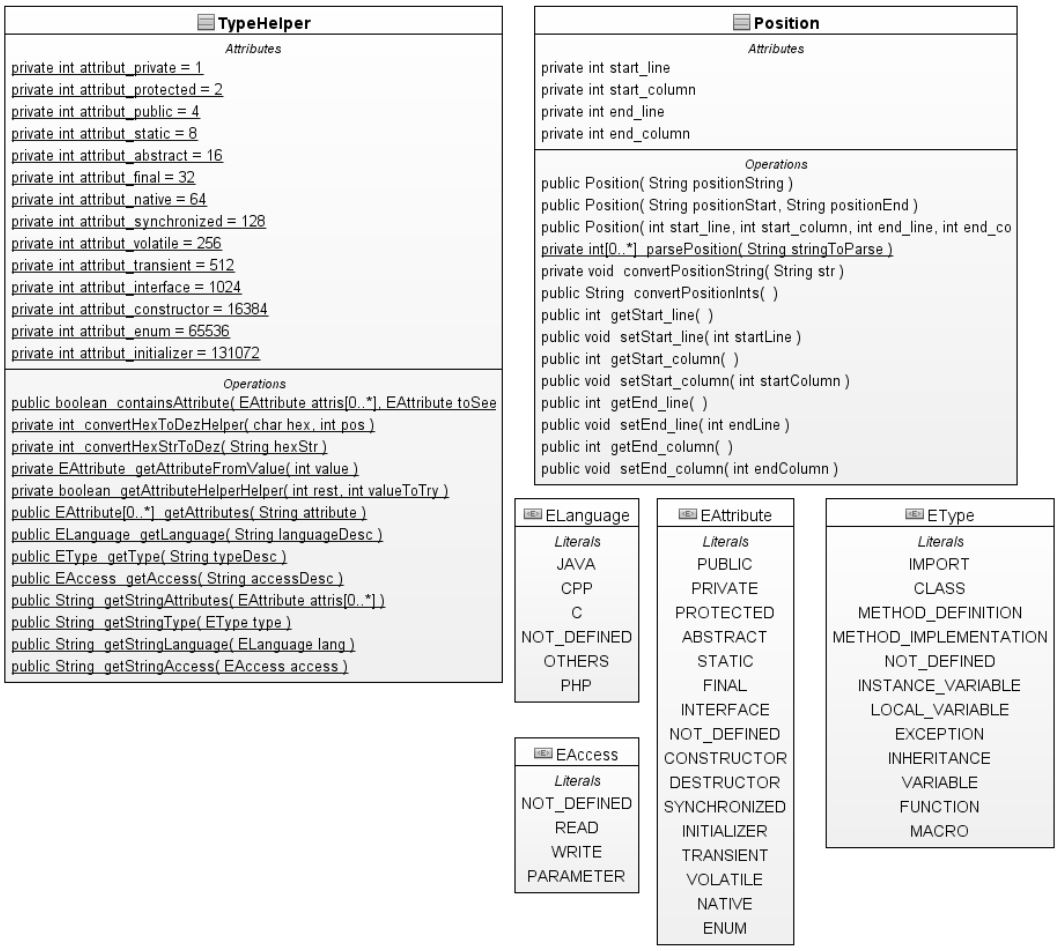

Figure 23: CacheTypes class diagram

B.3.3 Parser class diagram

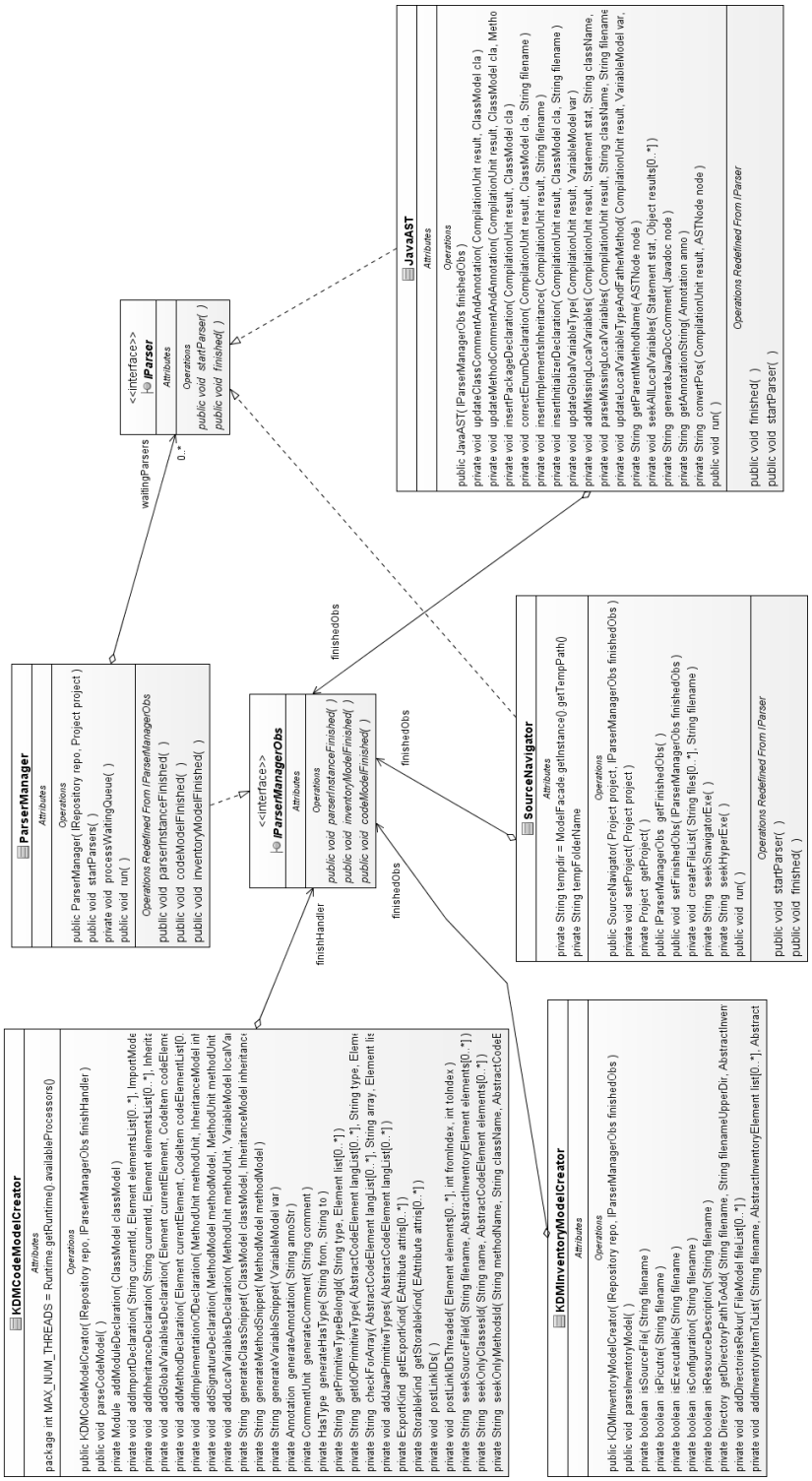

Figure 24: Parser class diagram

B.3.3.1 Parser Helper class diagram

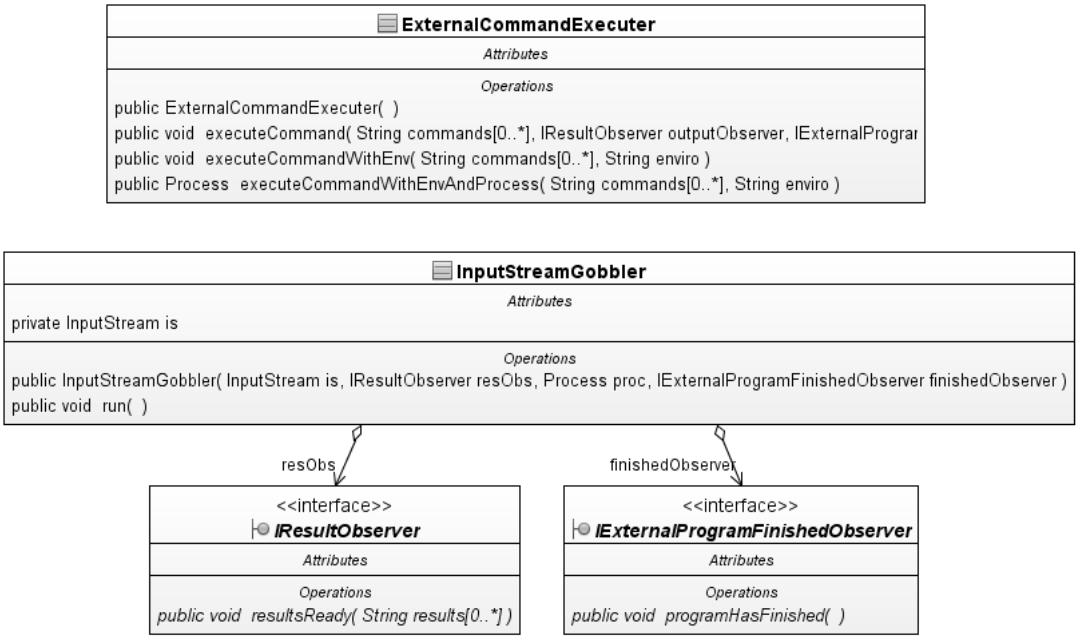

Figure 25: Parser Helper class diagram

B.3.3.2 SourceNav class diagram

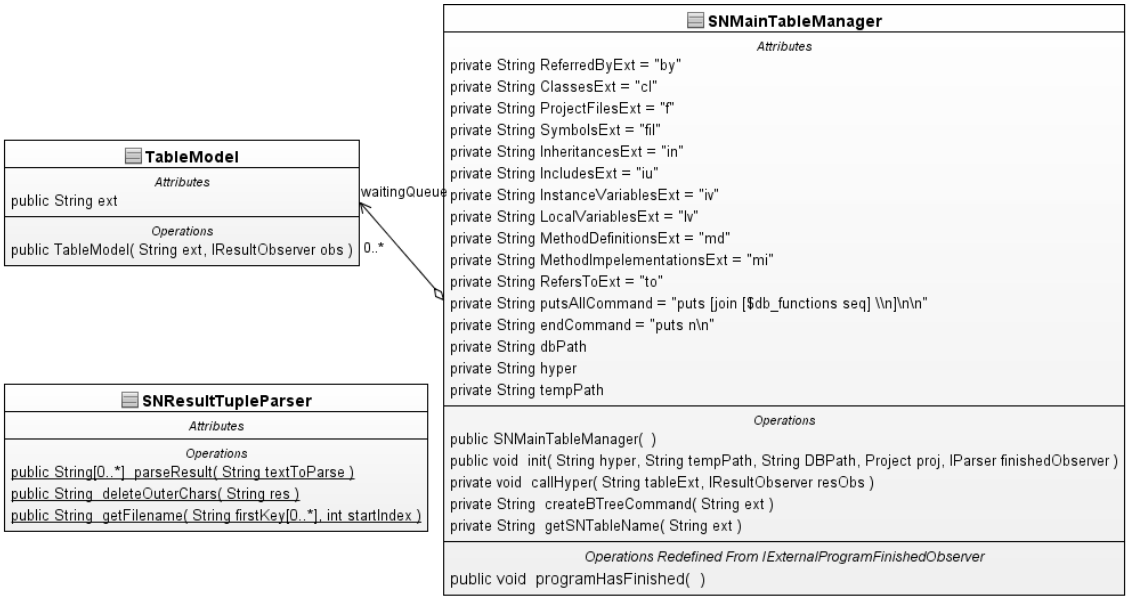

Figure 26: SourceNav class diagram

B.3.3.3 SourceNav Handler class diagram

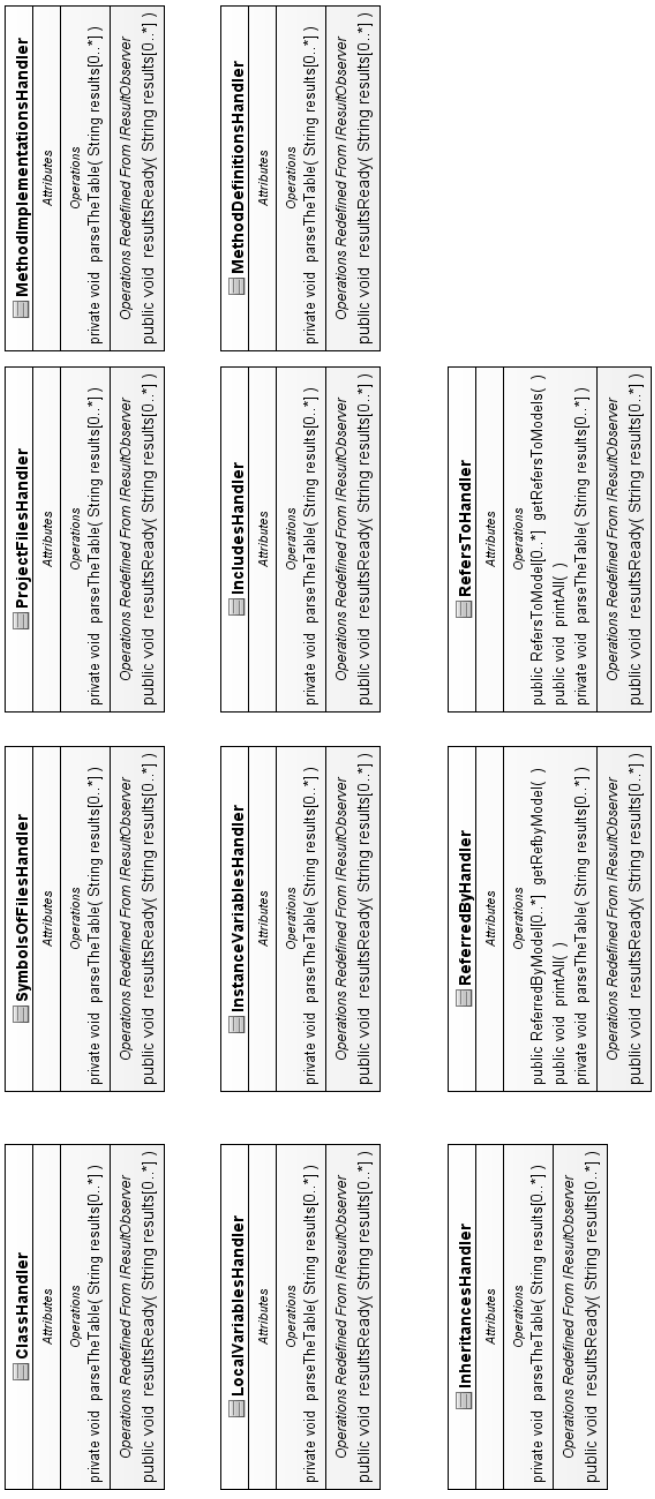

Figure 27: SourceNav Handler class diagram

B.3.3.4 SourceNav Models class diagram

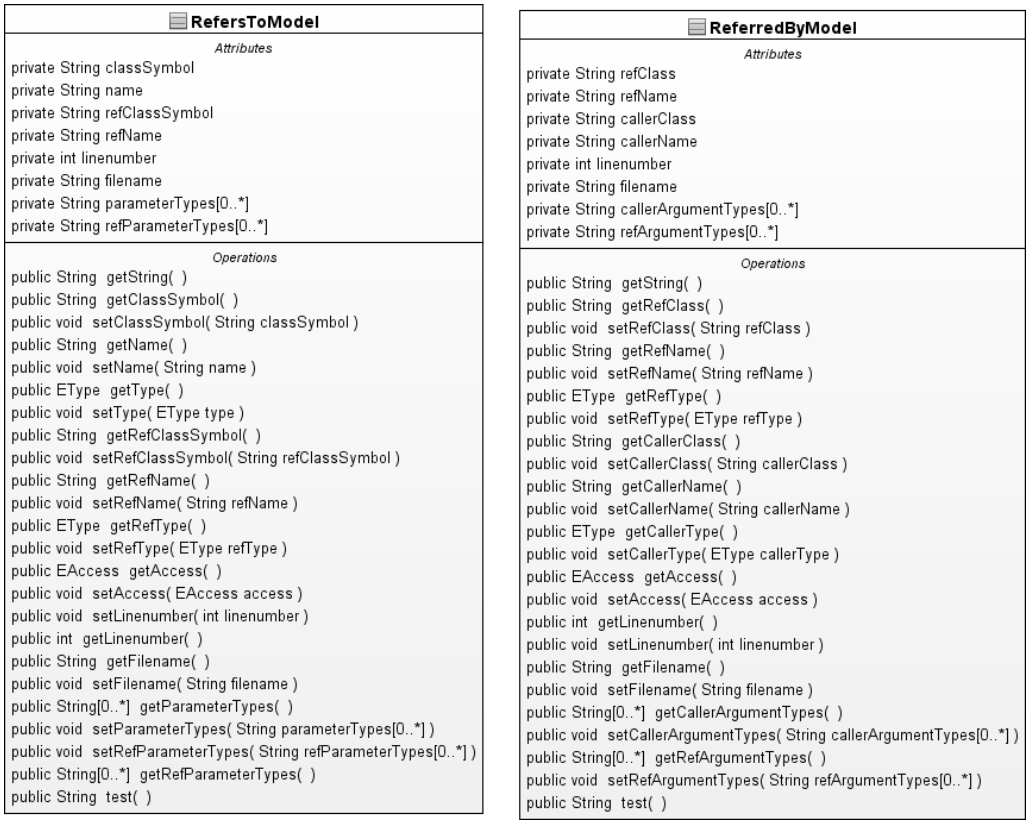

Figure 28: SourceNav Models class diagram

B.3.4 SAR Manager class diagram

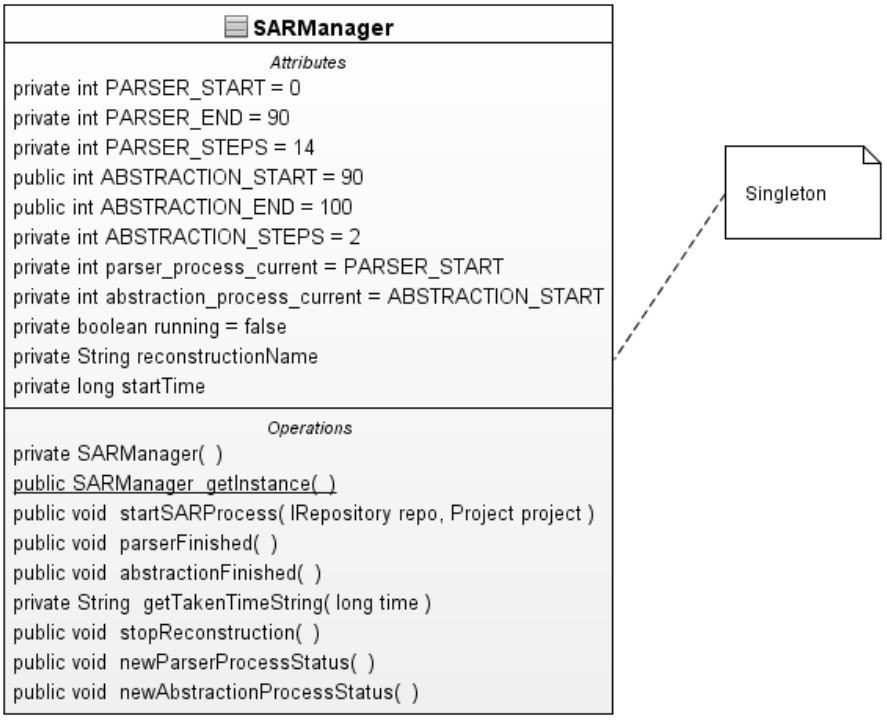

Figure 29: SAR Manager class diagram

B.3.5 Tools class diagram

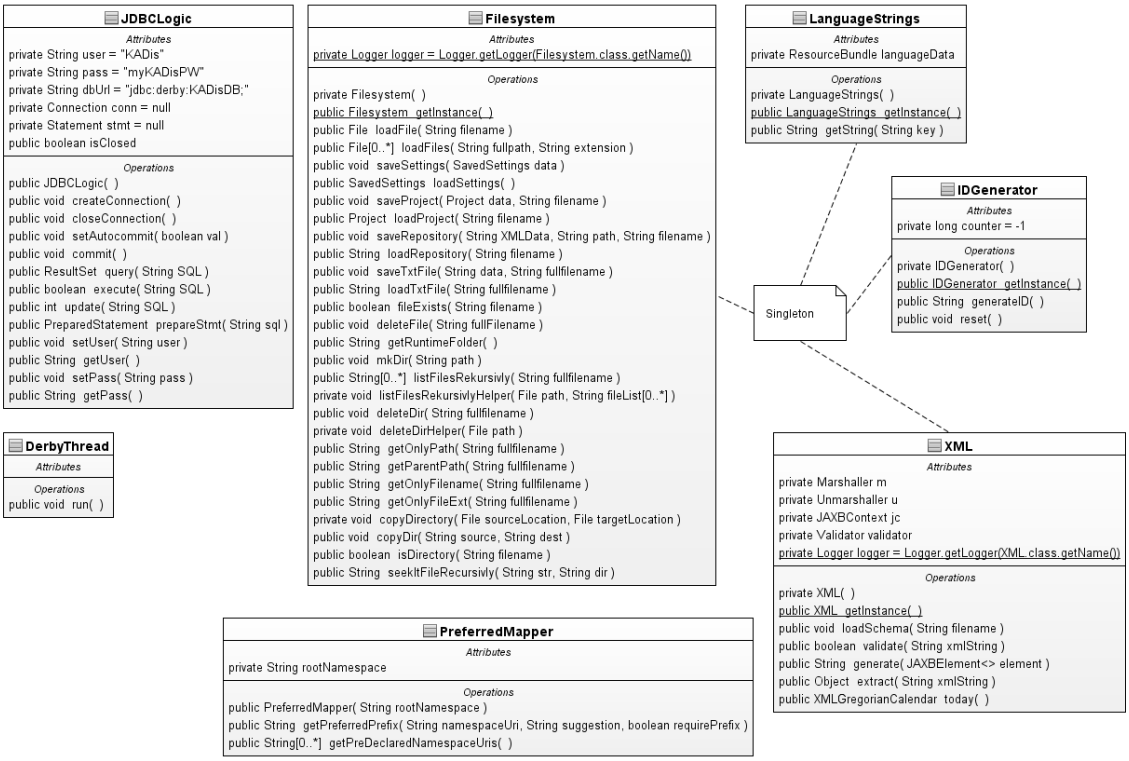

Figure 30: Tools class diagram

F Attachments

- One DVD labeled *Bachelorthesis attachment - Florian Fittkau* containing the source code for KADis, created documents in pdf-format, used external test applications, and generated KDM outputs for the used test applications