Kiel University, Kiel, Germany
Department of Computer Science
Software Engineering Group

Diploma Thesis

Transformation of Java Bytecode to KDM Models as a Foundation for Dependency Analysis

Oliver Prinz (opr@informatik.uni-kiel.de)

01. August 2012

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

Abstract

Maintenance is a key activity in the long-term support of software. Faulty parts have to be fixed and new requirements incorporated; the system evolves. But changing system parts involves program understanding; a task that is often connected with a reverse engineering step based on the system's code. A problem arises, if the source code is not available and has to be retrieved by a decompiler, for instance. For most programming languages, more than one exists, but they differ in their functionality, therefore a suitable decompiler has to be selected for this task. Embedded in this context is the first part of this thesis, where current available Java decompilers are evaluated; the best one will be integrated into *CloudMIG Xpress*, a tool supporting the migration of legacy software systems onto a cloud environment architecture. A core part of CloudMIG Xpress is the detection of *cloud environment constraint violations* — breaches of restrictions imposed by a cloud environment provider — that generates the context for the second part of this thesis. A dependency analysis library is introduced, that proposes a solution to the limitations of the constraint violation detection mechanism; he is not capable to determine the using context of such violations.

Eidesstattliche Erklärung

Hiermit erkläre ich, Oliver Prinz, geb. 03.09.1984, an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

Sleep is not a valid substitute for coffee.

– *Common student saying*

Contents

1. Introduction **15**
1.1. Motivation 15
1.2. Goals of this Thesis 18
1.2.1. Java Decompiler Integration 18
1.2.2. KDM-based Dependency Analysis 18
1.3. Structure of this Thesis 18

2. Foundations and Technologies **21**
2.1. Foundations 21
2.1.1. Reverse Engineering 21
2.1.1.1. Overview 21
2.1.1.2. Decompilation 22
2.1.1.3. Bytecode 24
2.1.2. Dependency Analysis 24
2.1.2.1. Transitive Dependencies and Transitive Closure 27
2.2. Relevant Technologies 28
2.2.1. Knowledge Discovery Meta-Model (KDM) 28
2.2.1.1. Infrastructure Layer 28
2.2.1.2. Programs Elements Layer 30
2.2.1.3. Runtime Resource Layer 32
2.2.1.4. Abstractions Layer 35
2.2.2. MoDisco 35
2.2.2.1. Model Discovery 36
2.2.2.2. Model Understanding 36
2.2.3. CloudMIG 37

3. Approach of this Thesis **41**
3.1. Reverse Engineering of Java Class Files 41
3.2. Dependency Analysis 42

4. Java Decompiler **45**
4.1. Overview 45
4.2. Available Java Decompilers 46
4.3. Test Cases 47
4.3.1. HelloWorld 48
4.3.2. Inner Class 48
4.3.3. Conditions 48
4.3.4. Exceptions 49
4.3.5. Loops 49

4.3.6. Inheritance . . . 49
4.3.7. Generics . . . 50
4.3.8. Annotations . . . 50
4.3.9. Enumerations . . . 50
4.4. Test Evaluation . . . 52
4.5. Discussion . . . 55
4.6. Integration into CloudMIG Xpress . . . 56

5. KDM-based Dependency Analysis 59
5.1. Overview . . . 59
5.2. Approach . . . 59
5.3. Implementation . . . 62
5.3.1. Nodes of the Closure Graph . . . 62
5.3.2. Edges of the Closure Graph . . . 64
5.3.3. Traversing a KDM Instance . . . 65

6. Evaluation 69
6.1. Overview . . . 69
6.2. Evaluation of the Dava Decompiler . . . 69
6.2.1. MyBatis JPetStore . . . 70
6.2.2. JForum . . . 70
6.2.3. Output Evaluation . . . 71
6.2.4. Discussion of the Evaluation . . . 75
6.3. Evaluation of the KDM-based Dependency Analysis . . . 75
6.3.1. Evaluation Tests . . . 77
6.3.2. Results . . . 79

7. Conclusion 81
7.1. Summary . . . 81
7.2. Future Work . . . 82

8. Related Work 83
8.1. Decompiler Analyisis . . . 83
8.2. Dependency Analysis for Program Understanding . . . 83

References 85

A. Test Implementations 91

B. Attached CD 101

List of Figures

1. The horseshoe model. It describes the steps that have to be taken in order to migrate an old software system to a modern architecture [24]. 16
2. Systems and their dependencies. 17
3. Extended transformation chain for the KDM trasformation process. 18
4. Common architecture for reverese engineering tools [9]. 22
5. Binary (in hexadecimal notation), assembly, and corresponding C code [8]. 23
6. The HelloWorld program in Java. 24
7. Java's HelloWorld converted to bytecode. Retrieved through *javap*. 25
8. The HelloWorld program in C. 25
9. C's HelloWorld converted to I-32 Assembler. 25
10. Dependency graph example. 26
11. Example of a Control flow graph for a given program. 27
12. Transitive closure graph from Figure 10. 27
13. Layers, packages, and separation of concerns in KDM [39]. 29
14. UML Class Diagram of the inventory model [39]. 30
15. Key classes of the Code package [36]. 31
16. Instantiation and assignment of the class "java.io.File" as KDM relationships. 33
17. The two phases of MoDisco: "Discovery" and "Understanding" [7]. 36
18. The CloudMIG approach. [18] 38
19. The transformation chain to a KDM instance in MoDisco. 41
20. Extended transformation chain. 42
21. Possible states of a system in the constraint detection process. [19] 43
22. Compilation and decompilation workflow for Java bytecode [21]. 45
23. Excerpt from ClassTest. 48
24. *Try-catch-finally* syntax in Java. 49
25. *For-Each Loop* in Java. 49
26. ArrayList signature with *Generics* in Java. 50
27. Day enumeration type, as defined in the Enum test case. 51
28. *Double Brace Initialization* pattern in Java. 51
29. Inline initialization of an abstract class in Java. 51
30. System to KDM transformation process in CloudMIG Xpress. 58
31. Discovery algorithm example for the transitive closure. 61
32. Double transitive edge. 62
33. Class diagram of *AbstractNode*. 63
34. Concrete nodes for the transitive closure graph. 63
35. Relationship-classes for the closure graph. 64
36. Interface of the dependency analysis library. 66
37. Entity-Handlers that are currently defined in the analysis library. 67
38. The *NodeManager* class diagram. 67

39. Exit condition example. 71
40. Loop example in JPetStore. 72
41. Missing *try-catch* statements. 72
42. Replaced For-Loop. 73
43. Class signature with interface implementation. 74
44. Generics and the decompilation counterpart. 74
45. Annotations in Dava 75
46. An enum type compared to the decompiler output. 76
47. Places in code, where prohibited classes can occur. 77
48. InheritanceTest class diagram. 78
49. Test class containing a member and a local variable. 78
50. Test classes for transitivity tests. 78

List of Tables

1. Test evaluation of Mocha. 52
2. Test evaluation of Jdec. 53
3. Test evaluation of JODE. 53
4. Test evaluation of Jad. 54
5. Test evaluation of Dava. 54
6. Test evaluation of JReversePro. 54
7. Test evaluation of Java Decompiler. 55
8. Test evaluation of Java Decompiler Project. 55
9. Test evaluation summary. 56
10. Decompilers support status and their license. 57
11. JPetStore Performance metrics. 70
12. JForum Performance metrics. 70
13. Tests passed by the analysis library. 79

Glossary

API	Application Programming Interface. An interface provided by a system to enable other systems the access to it.
AJAX	Asynchronous JavaScript and XML A group of techniques to create asynchronous web applications.
EMF	Eclipse Modeling Framework A framework to model and generate code.
GPL	GNU General Public License *http://www.gnu.org/licenses/*
J2EE	Java 2 Platform, Enterprise Edition Java computing platform for enterprise systems.
JAR	Java Archive Archive format similar to ZIP.
JDK	Java Development Kit Software development kit for Java programs.
JVM	Java Virtual Machine Runtime environment for Java bytecode.
XMI	XML Metadata Interchange Standard to exchange meta-data through XML.
XML	Extensible Markup Language A markup language for storage and transportation of data.
javac	A Java compiler included in the JDK.
javap	A Java class file disassembler included in the JDK.

1. Introduction

Ladies and gentlemen please
Would you bring your attention to me?
For a feast for your eyes to see
An explosion of catastrophe

– *Saliva, Ladies And Gentlemen*

1.1. Motivation

Long-term support of software is often affected by technology evolution. Operational requirements change and have to be adapted or completely new ones have to be developed and implemented throughout the life-cycle. Examples may be the integration of new functions and features, upgrade of used frameworks or the adjustment of API calls. The progress of hardware has to be addressed also, as the behavior changes or the availability of specialized parts decreases. Another aspect to consider is the rise of new technologies and the accompanied changes in computing paradigms, like the introduction of *Object-oriented Programming* (OOP). All this and more coins the term of *Software modernization*: A process of converting or porting a legacy system to state of the art technology.

Core activities of this process are *Reverse Engineering*, *Re-Engineering*, and *Forward Engineering*, defining an order on states a modernization goes through. They are derived from the "horseshoe" defined by Kazman et al., which is a "visual metaphor of the integration of code-level and architectural re-engineering views of the world," [24] and is figured in 1. The first activity addresses the generation of facts about a system that are useful for the process, such as the architecture or design patterns. These facts are then used in the re-engineering activity to restructure the relevant system parts according to the new requirements, which are implemented in the last activity.

In June 2003 the *Object Management Group*[1] formed a task force to build standards, that can be applied to the modernization of legacy systems. In applying concepts and standards of Model-Driven Architecture to the process of modernization, the task force proposed in November 2003 a specification called *Knowledge Discovery Meta-Model* (KDM); it describes an intermediate representation of existing software systems through a set of models. A key part of this specification is the ability of modeling a variety of systems in a uniform way, independent from the language they are implemented in.

[1] http://www.omg.org/ (accessed 12.01.2012)

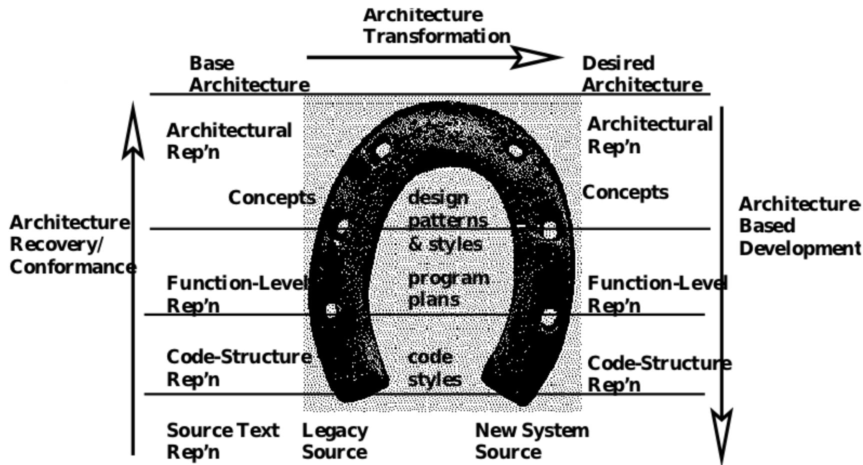

Figure 1: The horseshoe model. It describes the steps that have to be taken in order to migrate an old software system to a modern architecture [24].

Today, the target environment, that many legacy systems are migrated to, is called "the cloud". It describes a computing environment with the following characteristics [35]:

On-demand self-service The capability of supplying a consumer with computing capacity as needed and in an automatic way.

Broad network access Services are available though standard network mechanisms, allowing users the access with a broad set of devices.

Resource pooling Resources are provided through a multi-tenant model, which are dynamically assigned in order to satisfy current demands of consumers. Different systems may run on the same physical machine without the knowledge of the consumer, as location services on this level are often non-existent.

Rapid elasticity Resources are automatically assigned or withdrawn to scale with demand, appearing to be unlimited, since resources can be acquired at any size and at any time.

Measured service As resources scale with the demand, the cost of the usage has to be measured. This happens through different metrics appropriate for the given resource, that are also transparent for the consumer and cloud provider.

Furthermore, the environment comes in three different service models, called *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS), and *Software as a Service* (SaaS), describing three different types of resource *control*. IaaS provides a consumer with

basic computing resources, like storage and networks, where user-defined software can be deployed. PaaS restricts access to resources; it allows the deployment of applications only on a previously defined platform. The lowest form of control is defined by SaaS, where a consumer can only provide pre-defined applications by the provider of the environment.

A tool to ease the effort of porting systems to a cloud environment is *CloudMIG Xpress*. It is a research project aimed at a semi-automated process for transforming legacy software systems to an architecture suitable for running in a cloud environment. Core parts of this tool are concerned with the detection of *Cloud Environment Constraint Violations*: Breaches of restrictions imposed by a cloud provider on systems. These restrictions prohibit the use of certain technical actions, such as file access.

CloudMIG Xpress relies on the *MoDisco* framework to generate KDM instances of legacy systems, that will be ported to a cloud environment. Unfortunately, the framework is still in development and has therefore some shortcomings. While it has a build-in support for Java source and bytecode, the generated KDM instances from them differ in their powerfulness; the weaker is generated from bytecode. Here, only the signatures of methods are considered and the bodies are spared. However, the latter contain vital parts of the system and are therefore needed in the migration process.

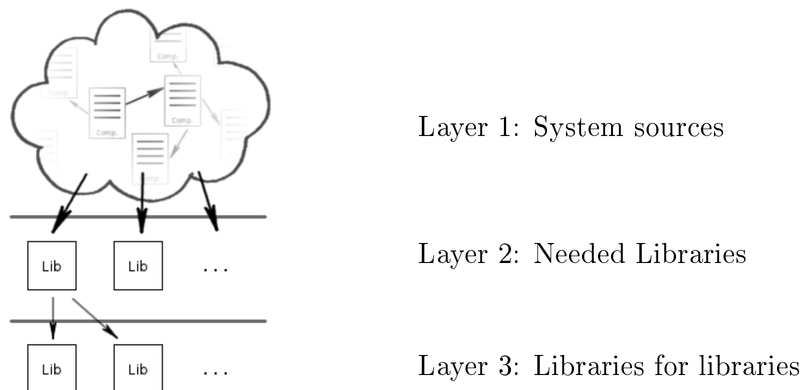

Figure 2: Systems and their dependencies.

Likewise, CloudMIG Xpress harbors the potential for improvement. While it can detect a variety of constraint violations, some of these detection mechanisms are insufficiently powerful, namely the detection of use of prohibited methods, functions and classes. While the occurrence check for such constructs is a rather easy task — it is simply a traversing of the KDM instance — the determination of usage is harder, since it involves the evaluation of the control flow inherent in the KDM instance.

Considering the example given in Figure 2, which shows the layers of dependencies in software systems. System parts depending on other system parts as well as on libraries,

which define solutions for common tasks. Those depending in turn on other libraries, and so on. If a constraint violation is detected, for example, in layer three, it is not obvious, if this violation affects the system in any way, since the violation could occur in a part of the library that is never used; the violation can be neglected.

1.2. Goals of this Thesis

This thesis has two main objectives. The first pursues a better support for Java bytecode in CloudMIG Xpress by integrating a decompiler, the second will extend the capabilities of the constraint violation detection mechanism.

1.2.1. Java Decompiler Integration

The proposed solution for a better Java bytecode support is depicted in Figure 3. The goal will be an extension of the transformation chain implemented in MoDisco with a decompiler. Because a variety of Java decompilers exists, an evaluation has to be done beforehand, to determine the best suited for this task.

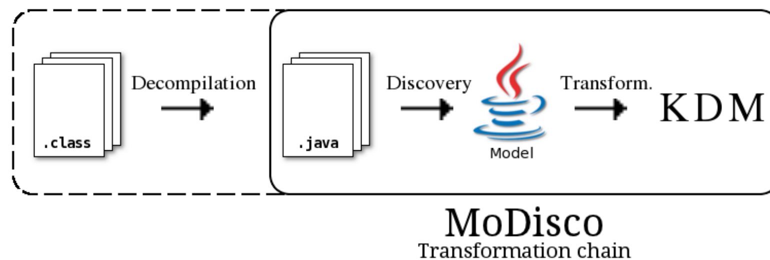

Figure 3: Extended transformation chain for the KDM trasformation process.

1.2.2. KDM-based Dependency Analysis

The usage of prohibited classes, functions, and methods can be determined through a dependency analysis. The proposed solution is the construction of the *transitive closure* from the KDM instance, containing all relevant entities and relationships defined in the different models of the instance. The traversal of the instance is realized through delegation; a handler object for each relevant entity takes care of the creation of nodes and edges in the resulting closure graph.

1.3. Structure of this Thesis

The remainder of this thesis is structured as follows: Section 2 describes the foundations and technologies used in this thesis, while Section 3 contains the detailed problem

description and taken approach for the proposed solutions. Section 4 evaluates current available Java decompilers, while Section 5 contains the implementation of a dependency analysis based on KDM. An evaluation of both approaches can be found in Section 6, the conclusion is given in Section 7. Section 8 closes this thesis with related work.

2. Foundations and Technologies

> Hamburgers! The cornerstone of any nutritious breakfast.
>
> – *Jules Winnfield, Pulp Fiction*

The following sections provide an overview of the foundations and relevant technologies used in this thesis. Section 2.1.1 describes the process of reverse engineering, whereas Section 2.1.2 discusses dependency analysis on given software artifacts. The last three sections deal with the used technologies in this thesis, describing a meta-model for modeling software systems from different architectural viewpoints named KDM in Section 2.2.1, a framework to discover an instance of the meta-model from given systems in Section 2.2.2 as well as a tool for migration of software systems in Section 2.2.3.

2.1. Foundations

2.1.1. Reverse Engineering

2.1.1.1. Overview

To enable the evolution of software, i.e. adapting and extending features as requirements change, it is crucial to understand the concepts implemented in a system. These concepts are mostly high-level, such as the chosen architecture for the system or implemented business rules and are often not directly apparent in the source code of the system [12; 44]. Thus, the consulting of the documentation is mandatory, but can lead to unsatisfactory results, if the system is only poorly, partial, or not at all documented. In such a case the information has to be retrieved from the code; it is the only reliable information source. This retrieving process is called *Reverse Engineering*. A precise definition is given by Chikofsky and Cross [9], whereby Reverse Engineering consists of the "identification of a system's components and their interrelationships" and "the creation of representations of a system in another form or at higher level of abstraction."

This process has its origin in the analysis of hardware, where it is used for "developing a set of specifications for a complex hardware system by an ordinary examination of specimens of that system." [41] The process tries to determine the purpose of a given hardware system and how it is achieved through analysis of the physical structure as well as an operation analysis.

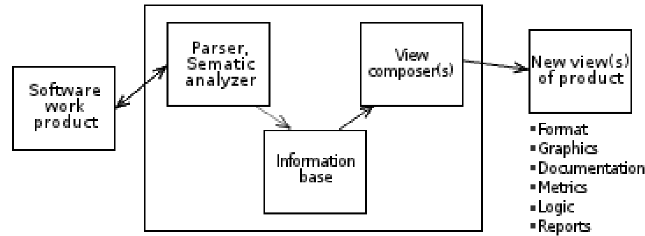

Figure 4: Common architecture for reverese engineering tools [9].

Eliam and Chikofsky show a wide application area [16], where this process can be applied. It can be grouped in the following manner:

Security-related: The process of reverse engineering is used in many different aspects of system or computer security. A system can be analyzed to exploit security flaws for malicious as well as for benevolent intends. An example would be the everlasting fight between the producers of so called "malware" and the developers of anti-virus software. In the field of cryptography, algorithms and their implementations are reverse engineered, to reason about the overall security they provide.

Software Development-related: Reverse engineering in the field of software development occurs often in the context of software maintenance. Developers can use the technique to determine the interoperability of partial or undocumented software or the quality of third-party code. The process can also be used to address other engineering problems, such as the recovering of a system's architecture [12; 3] and used design patterns [4; 38; 43], but also in migration processes towards new platforms [18], among others.

In order to cope with the complexity of systems in the reverse engineering process, automation is needed. A common architecture for tools, that automates the process, is shown in Figure 4. The system under view is analyzed and the results are put into an information base. View composers then generate different views from these information, giving different views on the system, like metrics, graphics or documentations.

An example tool implementing this architecture is *CodeSurfer* [2]. It uses static analysis techniques on C and C++ for review purposes. It comes with a set of code parsers, that are able to analyze pointers and dependencies between data functions, among others. Generated views contain mostly graphs, showing control flow or interdependencies.

2.1.1.2. Decompilation

As stated before, the lack of documentation leads to analyzing the source code of a system. A problem arises, when the system, or parts of it, are available only in binary

```
55 89 e5 8b
45 0c 03 45
08 01 05 00
00 00 00 89
ec 5d c3
```

```
sum:
  push1 %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  addl %eax, accum
  movl %ebp, %esp
  popl %ebp
  ret
```

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

Figure 5: Binary (in hexadecimal notation), assembly, and corresponding C code [8].

or executable form. This case often occurs in the context of legacy system maintenance, where the company that created the system does not exists anymore or where the data has accidentally been deleted and could not be recovered. In such a case the binary data has to be examined. However, reading and understanding binary code is hard work, therefore the data is most often transformed into a human readable form. This transformation process is in essence the inverse of compilation and is called *decompilation*: The process of converting binary data or an executable program back to a (high-level) programming language. Figure 5 shows an example for such a decompilation process. The binary data (for convenience given in hexadecimal notation) on the left will be transformed to an assembly language in the middle,[2] which in turn is translated into the higher programming language C on the right. It is quite obvious, that the process is time consuming and error-prone if done manually and in a larger scaled programs, consisting of millions of lines of code; especially the transformation from assembly to a higher language is apparently not trivial. That is why this process is often done with a *Decompiler*, a program, that automates the transformation.

A Decompiler takes one or more binary files as input and convert them to the target language determined by the program. Unfortunately, this conversion process has to overcome many obstacles [11]:

Data and instructions representation: This problem derives from the Von Neumann architecture, where data and instructions are indistinguishable.

Added Subroutines: The compiler and linker add a great number of subroutines for environment setup and runtime support. Because they are mostly written in assembler, they are often not translatable to higher-level functions.

Self-containment of binary programs: Executables are often self-contained; stand-alone programs, where relevant libraries of the operating system are bound into the binaries.

[2] This process is often called disassembly; it translates machine language into assembly language.

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

Figure 6: The HelloWorld program in Java.

Finally, the original names of functions, variables, etc. as well as some syntactic structures cannot be retrieved through decompilation — because they are discarded by the compilation process — creating a merely functional equivalent to the original.

2.1.1.3. Bytecode

A special form of compilation output is bytecode. It is an intermediate representation for programs intended to run on an *interpreter*,[3] a program which executes bytecode. It has a human readable form, much like an assembly language, and is often enriched with some sort of meta-data, aiding the reasoning about the original purpose of the code. Figure 6 shows the prominent Hello World program written in Java with the corresponding bytecode in Figure 7. Compared to the assembly code in Figure 9 created from the C HelloWorld in Figure 8, it can be seen easily, that the bytecode is much more verbose, containing method names and method parameter values.

The capability of easily porting bytecode to other systems is an often cited advantage. In general, if a program is written and compiled for one kind of platform, it cannot run easily on another, because of compatibility problems: Each platform could describe its services in its own unique way or use a different instruction set. In using bytecode, the program can run on every platform that is supported by the interpreter.[4]

A common disadvantage associated with bytecode is the *Execution Time*. Before the bytecode can be executed, it has to be transformed by the interpreter, adding time to the overall runtime of the program. Indeed, this was true in the past, the introduction of so called *JIT*[5] *Compilers* renders this statement today mostly false [1; 20]. While there is still a performance loss in the execution time, it has become less significant.

2.1.2. Dependency Analysis

A dependency is commonly defined as “the quality or state of needing something or someone,” [30] thus the term dependency analysis refers to the process of identifying

[3] Often called a *Virtual Machine* (VM)

[4] In Theory. In practice, this is sometimes not the case. A famous example is the Standard Widget Toolkit for Java programs. For each platform (like Linux and Windows), there is a different Toolkit, making the program using it, platform depend.

[5] Just-in-Time

```
Compiled from "HelloWorld.java"
public class HelloWorld extends java.lang.Object{
public HelloWorld();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   getstatic   #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   3:   ldc      #3; //String Hello World!
   5:   invokevirtual   #4; //Method java/io/PrintStream.println:
                             (Ljava/lang/String;)V
   8:   return

}
```

Figure 7: Java's HelloWorld converted to bytecode. Retrieved through *javap*.

```c
#include <stdio.h>
int main() {
    (void)printf("Hello World!\n");
    return 0;
}
```

Figure 8: The HelloWorld program in C.

```
.LC0:
    .string     "Hello World!"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    $.LC0, %edi
    call    puts
    movl    $0, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

Figure 9: C's HelloWorld converted to I-32 Assembler.

entities and the relationships between them, such that the relationships comply to the definition. More formally: Let S be a set of entities, $s, s' \in S$ and $R \subseteq S \times S$. A dependency between s and s' exists, if $(s, s') \in R$. Thus, the result of a dependency analysis is the pair (V, E), where $V \subseteq S$ defining the found entities and $E = \{(v, v') \in R : v, v' \in V\}$. Obviously, this pair represents a graph, where each node in the graph represents an identified entity and each edge a dependency, respectively. An edge is also directed, indicating that an entity depends on the entity on the end of the edge. Figure 10 shows an example of such a dependency graph. The nodes are the set of A, B, C, D, E, F, where D depends on E and F, B on C and A on D and B, thus creating the set of edges: (D,E), (D,F), (B,C), (A,D), (A,B).

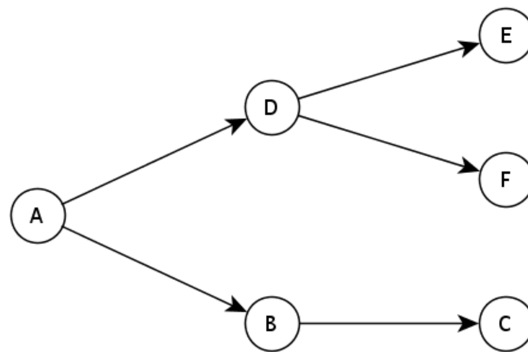

Figure 10: Dependency graph example.

In computer science, the dependency analysis is used in a variety of disciplines. For example in compiler theory, the analysis is used to determine the execution order of statements and to find circular dependencies. Also, the output can be used to reason about the safe reordering or the parallel execution of statements. In the field of software engineering it can be used as a pre-step for remodeling systems and to reduce complexity.

The *Control flow* and the *Data flow* Analysis are the two well known dependency analyses. The Control flow Analysis identifies the order, in which instructions in an analyzed system are executed or evaluated, respectively and is visualized through a *Control flow graph*. Figure 11 shows an example program and the resulting graph of a control flow analysis applied to this program.

A *Data flow Analysis* discovers dependencies between data elements, that are altered by the analyzed system. It determines the order of execution of statements through the data dependencies. For example, considering the assignments: x = a + b, b = 1 + 2, a = 5 + 3. The result of the analysis would show, that the assignments of a and b have to be evaluated before the assignment of x. The evaluation order of those statements is not further defined, since they do not share a dependency, thus they could be evaluated in any order or in parallel.

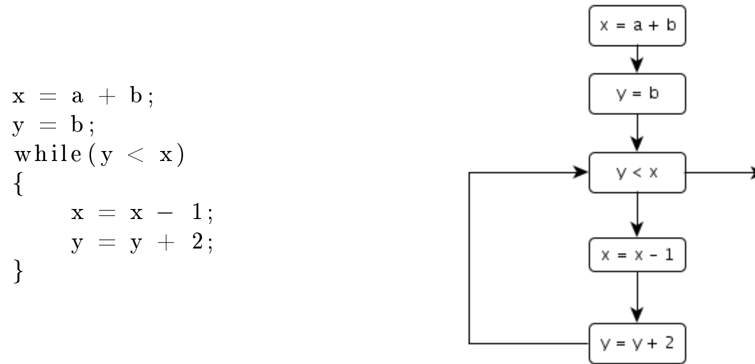

Figure 11: Example of a Control flow graph for a given program.

2.1.2.1. Transitive Dependencies and Transitive Closure

A transitive dependency between two entities exists, if there is a path in the dependency graph from one entity to the other. In graph theory, a path is a sequence of nodes, such that for each consecutive pair of nodes in the sequence, an edge between these nodes exists. More precisely: if $a, b, c \in E$ and $(a, b) \in R \wedge (b, c) \in R \rightarrow (a, c) \in R$. Figure 10 shows a dependency graph that includes three such transitive dependencies: A and E, A and F, and A and C.

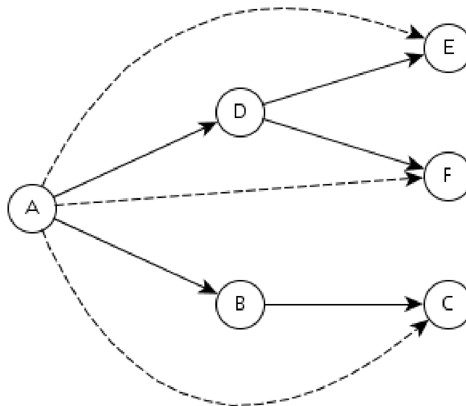

Figure 12: Transitive closure graph from Figure 10.

In a more complex graph such dependencies are hard to spot, that's why the *transitive closure* of the dependency relation is computed. The transitive closure is the relation R^+

over R, such that R^+ is a minimal transitive relation and contains R. Simply put, this closure contains all dependencies between entities, that are either direct or transitive. Figure 12 shows the transitive closure graph from the example given in Figure 10. The transitive edges are represented through a dotted line.

The closure is often needed to answer reachability questions; a node m is reachable from a node n, if they are connected through a path. The advantage over a normal graph is performance: In a normal graph the time needed to find the answer is $O(|V| + |E|)$ in the worst case (when a *Depth-first Search* or *Breadth-first Search* is used). With the transitive closure it is either $O(|V|)$, when implemented naively, or $O(1)$, when the information is saved in a $|V|\times 2$ matrix.

2.2. Relevant Technologies

2.2.1. Knowledge Discovery Meta-Model (KDM)

The *Knowledge Discovery Meta-Model* is a specification by the ADM Task Force[6] and specifies a "meta-model for representing *existing software*, its elements, associations, and operational environments." [36] It is the result of the efforts undertaken by the Task Force to standardize the re-engineering process in the context of software modernization by using model-driven principles.

The main goal of the KDM standard is the creation of a uniform representation of all software artifacts an existing system is composed of, with the aim to extract different kinds of knowledge about the system. A software artifact thereby refers to everything that is involved in the engineering process, such as the source code of the system, used databases, user interfaces or build instructions.

The extracted knowledge is composed of a set of facts, describing behavior, structure and data of the system. The facts can be grouped into domains, where each domain defines an architectural viewpoint of the system. The language used to describe such a viewpoint is contained in a package. Figure 13 shows all packages defined by the KDM standard. For example, the Data package defines the viewpoint for the Data domain, containing meta-model elements describing the organization of data in the system, like database tables. The packages are also structured in layers, each builds upon the previous, to cope with the complexity of KDM; they will be described in the following subsections.

2.2.1.1. Infrastructure Layer

A small set of concepts that are used systematically throughout the specification are defined in this layer, consisting of the three packages *Core*, *KDM* and *Source*.

[6] Architecture-Driven Modernization Task Force *http://adm.omg.org/* (accessed 12.01.2012)

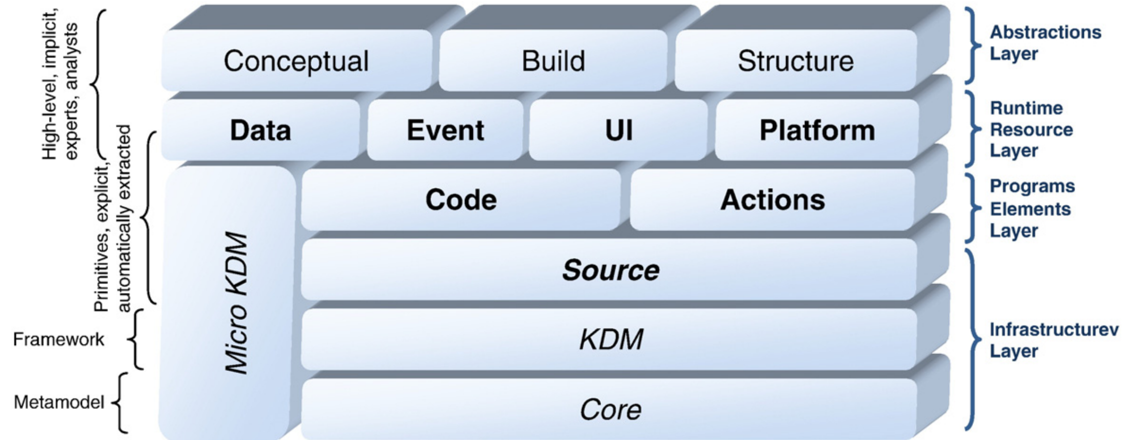

Figure 13: Layers, packages, and separation of concerns in KDM [39].

Core The *Core* package contains basic constructs for creating and describing the meta-model elements in all other packages. KDM builds on the Entity-Relationship terminology, thus the two fundamental classes are *KDMEntity* and *KDMRelationship*, from which every other element in the KDM specification derives. An instance of KDMEntity is thereby defined as "an abstraction of some element of an existing software system, that has a distinct, separate existence, a self-contained piece of data that can be referenced as a unit," [36] while an instance of KDMRelationship represents an association with a meaning between elements of the system.

KDM The package named *KDM*[7] contains meta-model elements that constitute the *framework* of each KDM instance. The framework describes the physical structure of the instance, which consists of one or more *Segments* containing one or more *KDMModels*. Each KDM package, that builds on this package, may define some specific KDMModel, addressing a specific facet of knowledge concerning the modeled system.

Source The Source package contains the *InventoryModel*. It is a collection of KDM-Entities that describe all the physical artifacts of the modeled system, such as source code, configuration files, images, etc. The model also contains a mechanism to trace a KDMEntity back to its "original" language-dependent representation, i.e., a reference or link to the (region of) a file, where the entity is defined. For example, a KDMEntity describing a function in a module contains a link to the lines of code in the source code file where the function is defined.

[7] To clarify the wording: There is a difference between the package named *KDM* and a KDM Package. The former describes the package of the infrastructure layer, while the latter is used for all packages of the specification.

Figure 14 shows the UML Class Diagram of the InventoryModel. For most physical artifacts there exists a KDMEntity, describing the type of the file. To address the grouping of artifacts, the package provides an *InventoryContainer* which can contain instances of *InventoryItems*.

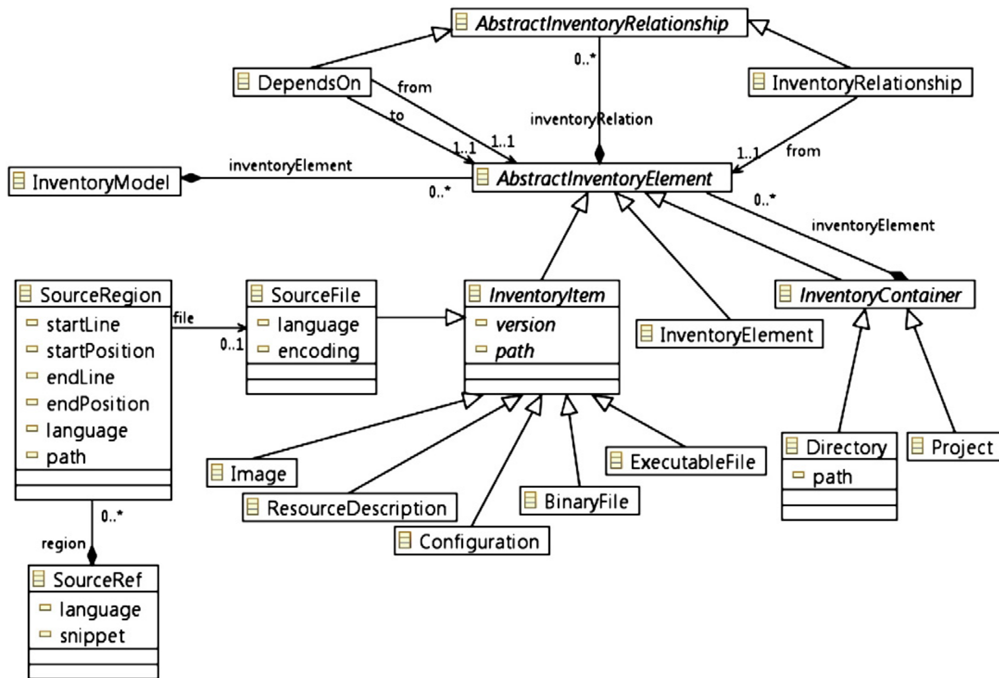

Figure 14: UML Class Diagram of the inventory model [39].

The package also provides an *AbstractInventoryRelationship* to model specific dependencies between physical artifacts. While the sub-class *InventoryRelationship* describes a general relationship between two inventory items, like the association between a source file and its compilation unit, the *DependsOn*-Relationship is used to model the requirement of one or more inventory elements in the build process of another inventory element, for example, the need of a certain configuration file for an executable file.

2.2.1.2. Programs Elements Layer

The *Program Elements Layer* consists of the packages *Code* and *Actions*. Various constructs from different programming languages can be represented through the meta-model elements defined in this package.

Code The *Code* package defines elements called *CodeItems*, representing "named elements determined by the programming language, the so-called 'symbols', 'definitions', etc.," [36] such as classes, and methods. Furthermore, the package defines elements to represent structural relationships between CodeItems.

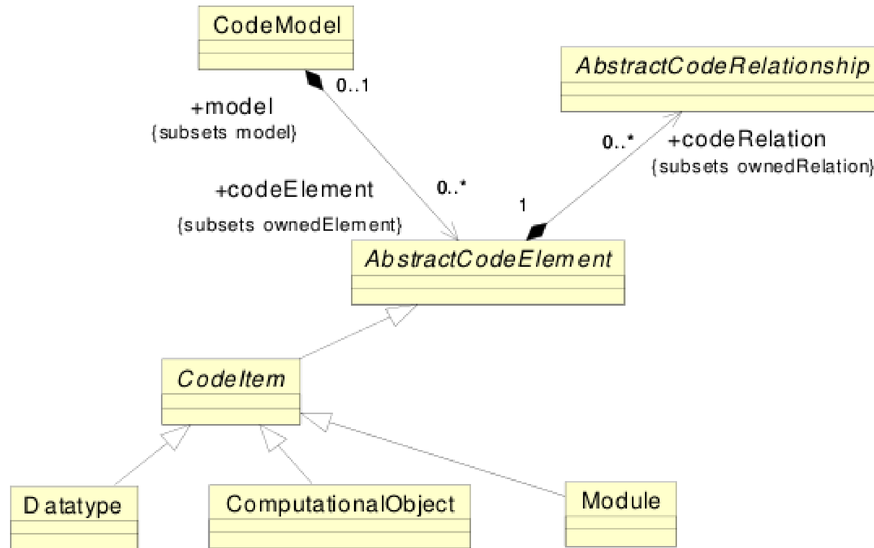

Figure 15: Key classes of the Code package [36].

Figure 15 shows the key classes of the Code package. It defines a *CodeModel* which owns a set of facts about the system modeled, as well as an *AbstractCodeElement* and an *AbstractCodeRelationship*. The latter is used to represent relationships between code elements, such as an import or extends relationship, which can be found in some object oriented programming languages, for instance.

The *AbstractCodeElement* is the abstract parent class for entities that can be used to model code. Derived from it is the CodeItem, an abstract class to constrain the owning relationship of some containers used in the Code package. Sub-classes thereof are used to model actual code elements:

- *Module*: A Module handles packaging aspects of code, like packages or compilation units. It represents an "entire software module or a component" [36], which is a "discrete and identifiable program unit that contains other program elements." [36]
- *ComputationalObject*: A generic object representing elements that are *callable*, such as procedures or variables.
- *Datatype*: A Datatype and the derived sub-classes define "the named elements [. . .] that describes datatypes." [36]

Actions The *Actions* package extends the Code package and contains meta-model elements to describe implementation-level behavior as well as data and control flow relationships between the code elements. It does so by defining two key elements: The *ActionElement* and the *AbstractActionRelationship*.

The *ActionElement* describes behavior, such as statements and operators. The attribute *kind* specifies the precise semantics, defined in the *Micro KDM* package. Furthermore, to trace an *ActionElement* back to its origin in the source code, the *SourceRef* attribute is provided.

AbstractActionRelationship is an abstract parent class for relationships, that describe some sort of behavior. According to [39], the relationship can either represent a control flow between two actions (Defined by *ControlFlow* and its sub-classes) or an association originating in an action element and a code element.

Figure 16 shows an example of how the instantiation and assignment of Java's File class is translated into KDM Code and Actions entities. The local variable "file" is translated to an instance of *StorableUnit* from the Code package. The instantiation and assignment is represented through the relationships *Calls* and *Creates*, respectively. For clarity and readability, names and references have been modified.

For more basic data handling, the Actions package provides the *Reads*, *Writes* and *Addresses* relationships. While *Reads* and *Writes* are used to define a data flow from and to a data element, the *Addresses* is used to model the access of data inside a more complex data structure, such as the *struct* element in the C programming language.

2.2.1.3. Runtime Resource Layer

The *Runtime Resource Layer* describes knowledge about a modeled system and the environment it operates in. Elements of this layer represent resources provided by the runtime platform and "resource actions" to manage such resources, respectively. Each package of the Runtime Resource Layer defines its own entities and resources to describe specific concerns in the modeled system. For example, the *Event* package provides classes to model state and state transitions triggered by an event. Also, each package defines some structural relationships between resources, as well as specific actions relating to the manipulation of such resources through API calls.

The Runtime Resource Layer consists of the four packages *Data*, *Event*, *UI*, and *Platform*, each with their own respective KDMModels named after the package they reside in.

Data The *Data* package defines meta-model elements addressing the organizational aspect of data. In contrast to the Code package, which defines data as "application data", such as I/O variables and function parameters, data is referred here under the persistence aspect, resulting in elements that can model complex data repositories.

Since the Data package is particularly designed to model relational databases, it comes

Instantiation and assignment of Java's File class.

```
File file = new File("path to file");
```

KDM representation of above actions.

```
<codeElement
    xsi:type="code:StorableUnit"
    name="file"
    type="java.io.File" kind="local">
    <codeRelation
        xsi:type="code:HasValue"
        to="ActionElement:class instance creation"
        from="CodeElement:StorableUnit:file"/>
</codeElement>
<codeElement
    xsi:type="action:ActionElement"
    name="class instance creation"
    kind="class instance creation">
        <codeElement
            xsi:type="code:Value" name="string literal"
            type="java.lang.String" ext="&quot;path/to/file&quot;">
    </codeElement>
    <actionRelation
        xsi:type="action:Calls"
        to="java.io.File.File(String fileName)"
        from="ActionElement:class instance creation"/>
    <actionRelation
        xsi:type="action:Creates"
        to="java.io.File"
        from="ActionElement:class instance creation"/>
</codeElement>
```

Figure 16: Instantiation and assignment of the class "java.io.File" as KDM relationships.

with a broad set of elements to map the facts contained within such a database into the Data domain defined by this package. For instance, the abstract *DataResource* class is specialized into *DataSegment*, *RelationalTable*, *RelationalView*, *RecordFile* and so on, to give a fine grained view on the modeled data and its organization.

The package also delivers a new set of actions, derived from the AbstractActionRelatonship defined in Actions package:

- *ReadColumnSet*: Used to model data flow from the data resource, for example a SQL *Select* query.
- *WritesColumnSet*: Models a data flow to the data resource, like a *Insert* query.

- *ManageData*: For operations, that neither has a data flow from, nor a data flow to the data resource, but nonetheless represent data access, this meta-model element is used.
- *HasContent*: A structural relationship defining an interconnection between an action element and the data resource.

Event The *Event* package facilitates the modeling of high-level behavior of a system, notably event-driven state transitions. The elements of this package allows the modeling of two kinds of states: concrete states of specific state-machine languages, such as CHILL [48] and abstract states, like states in algorithms or user interfaces.

UI Meta-model elements in the *UI* package are related to facets of user interface knowledge, such as the composition of the interfaces, their sequence of operations, as well as their relationships to the modeled system.

The model of the UI package is the *UIModel*, which represents the main components of the user interface. The model is composed of *UIResource* elements, describing concrete ui elements such as forms or labels, and containers, like a web page, respectively. The package also defines two relationships, the *UILayout* and the *UIFlow*, where the former is used to describe an association between “a portion of a user interface and its layout” [39], while the latter “captures the behavior of the user interface as the sequential flow from one instance of Display to another.” [36]

Deriving from the Actions package AbstractActionRelationship, the UI package defines three new relationships, called *ManageUI*, *ReadsUI* and *WritesUI*, with similar semantics as the counterparts in the Action package, differing in the fact, that they refer to *UIResource* elements.

Platform The *Platform* package defines meta-model elements representing parts of the runtime platform and the environment of the modeled system, like inter-process communication or data management. These elements constitute the execution context of the system due to the fact that a system is not self-contained, meaning that the system is not only determined by the programming language it was written in, but also by the selected runtime platform.

The package defines the *PlatformModel*, a container for a set of *AbstractPlatformElements*. Sub-classes of the AbstractPlatformElement specify the type of platform or environment resource, like a Thread, a Process or a File-Resource. Also defined is the relationship *BindTo*, that describes an association between two platform resources.

For accessing such resources, the package provides also a set of actions, deriving from the Actions package. These actions are *ReadsResource*, *WritesResource*, *ManagesResource* with similar semantics as their counterpart from the Actions package.

2.2.1.4. Abstractions Layer

The *Abstractions Layer* contains implicit knowledge of a system, like domain- or application-specific abstractions, and will mostly not be generated automatically but by a user reasoning about or interpreting the system. To model this knowledge, the Abstractions Layer defines the three packages *Conceptual*, *Build*, and *Structure*.

Conceptual The *Conceptual* package is used to model domain-specific knowledge of the system; it describes the viewpoint for the domain of Business Rules. The elements in this package are used to generate conceptual information about the system by using lower level KDM models.

Furthermore, the resulting model is aligned with the SBVR[8] specification, where the entities defined in this package are mappings of the so called "concepts" of SBVR.

Build The *Build* package defines elements representing facts related to the build process of the modeled system, "including but not limited to the engineering transformations of the 'source code' to 'executables"' [36], as well as elements to represent the output artifacts generated by the build process.

Structure The *Structure* package defines several meta-model elements to describe the architectural components of the modeled system, such as subsystems, layers, packages, etc. The results show, how the structural elements of the system are related to the modules defined by the Code package.

The package defines the *StructureModel*, which owns a collection of *StructuralElements*, representing the organization of the modeled system's code.

Micro KDM This package is concerned with ActionElements. The problem addressed is, that some ActionElements have not per se a defined semantic. For views, that can be used in a static analysis, this ought to be an issue. Thus, constraints and attributes are introduced with this package, specifying additional semantics for such cases.

2.2.2. MoDisco

MoDisco is "a generic and extensible framework for model driven reverse engineering." [7] It is embedded in the context of reverse and re-engineering of software systems and applies the model-driven engineering principle to this process.

The main problem MoDisco tackles, is the cost of migration and modernization of legacy systems. Quick re-engineering of software systems to adapt the changing and extended demands to it, is often not possible, because the systems are heterogeneous

[8]Semantics of Business Vocabulary and Business Rules, *http://www.omg.org/spec/SBVR/* (accessed 12.01.2012)

and important for the business processes, which causes a deep integration in to the company's routine procedures. Combined with the fast pace of evolution of technologies, making developed systems rapidly obsolete, most companies get into a predicament.

The MoDisco framework tries to solve these problems by "switching from the heterogeneous world of implementation technologies to the homogeneous world of models." [7] The major challenge identified, is the discovery and understanding of functionalities, architecture, data, etc. used in the legacy systems and "reverse engineer them into a meaningful representation that can be later on manipulated and re-implemented." [7]

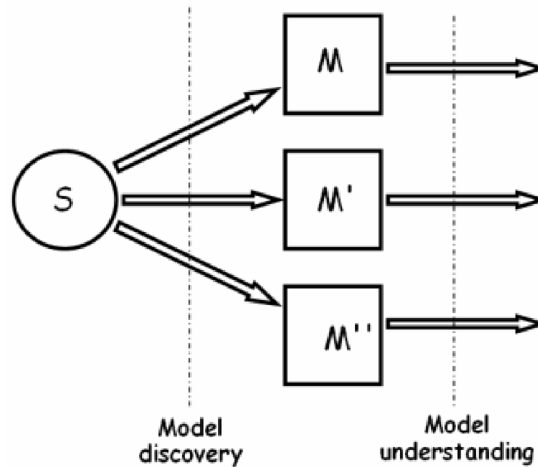

Figure 17: The two phases of MoDisco: "Discovery" and "Understanding" [7].

Figure 17 shows the approach taken by MoDisco, which consists of the two phases "Model Discovery" and "Model Understanding".

2.2.2.1. Model Discovery In this phase a set of models (denoted as M, M', M" in the figure) is discovered through one or more "discoverers", describing the different viewpoints needed, each expressed through a meta-model. The discovery process thereby considers not only the source code of the system under view, but also documentation, raw data, and other sources needed in the engineering process of the system. The output models can be instances of a variety of meta-models most notably the *Knowledge Discovery Meta-Model* (see section 2.2.1) and *Software Metrics Meta-Model*, both specifications of the *OMG*.

2.2.2.2. Model Understanding In this phase, the generated models can be exploited. In the context of reverse engineering, the models can be used for metrics computation, like coupling, refactoring of architecture and structures, code generation, quality anal-

ysis, and so on. While the content of the models is analyzed and computed, model transformations or chains of such transformations can occur, which often are automated processes, depending on the identified scenarios.

MoDisco is available as an Eclipse open source project, providing an extensible and customizable Model Driven Reverse Engineering (MDRE) framework. It is equipped with both generic and customizable components, such as a model browser, a mechanism to extend and customize models, a query manager for models, and some metrics visualization facilities. Also, it offers support for Java reverse engineering, including a Java meta-model, a corresponding discoverer and a transformation process to KDM, as well as XML reverse engineering.

2.2.3. CloudMIG

CloudMIG describes a model-based approach "for the semi-automated migration of enterprise software systems to scalable and resource-efficient PaaS- and IaaS-based applications." [18] The main purpose of the approach is to aid the process of re-engineering a legacy software system to run in a cloud computing environment, because most legacy systems have to be heavily re-engineered, to create an architecture suitable for running "in the cloud." However, the cause for this is mostly not the migration to a server-based structure, but the scalability and "elasticity" of the target environment.

While other approaches and projects related to such types of migration exist, they all suffer several shortcomings, summarized as follows [18]:

1. Applicability: Solutions for a cloud migration process are tied to a particular cloud provider.
2. Level of automation: The migration process is not fully automated; users often have to build the target architecture and the mapping model[9] manually. Also, the proposed solutions lack a detection mechanism for constraint violations of the cloud environment at design time.
3. Resource efficiency: The use of resources in migrated systems is often inefficient and the advantages of the environments elasticity aren't exploited. Also, the insufficient evaluation of dynamic resource utilization of most solutions adds to this shortcoming.
4. Scalability: Although one of the advantages of a cloud environment, an automatic evaluation of the scalability of a target architecture is often non existent.

[9] A model to describe, how elements of the legacy system are mapped to the elements of the new system.

The CloudMIG approach addresses these shortcomings by defining six activities for a migration of systems to PaaS- and IaaS-based cloud environments, using a model-based approach. The interaction of these activities can be seen in Figure 18 and are specified as follows:

A1 - Extraction: The legacy system is transformed into a set of models, describing the actual architecture of the system. The process is controlled by the MoDisco framework (see section 2.2.2), using OMG's KDM (see section 2.2.1) as the meta-model. Furthermore, a utilization model is extracted that includes statistical properties concerning user behavior, such as the number of service invocation over time and which also contains "application-inherent information related to proportional resource consumption." [18]

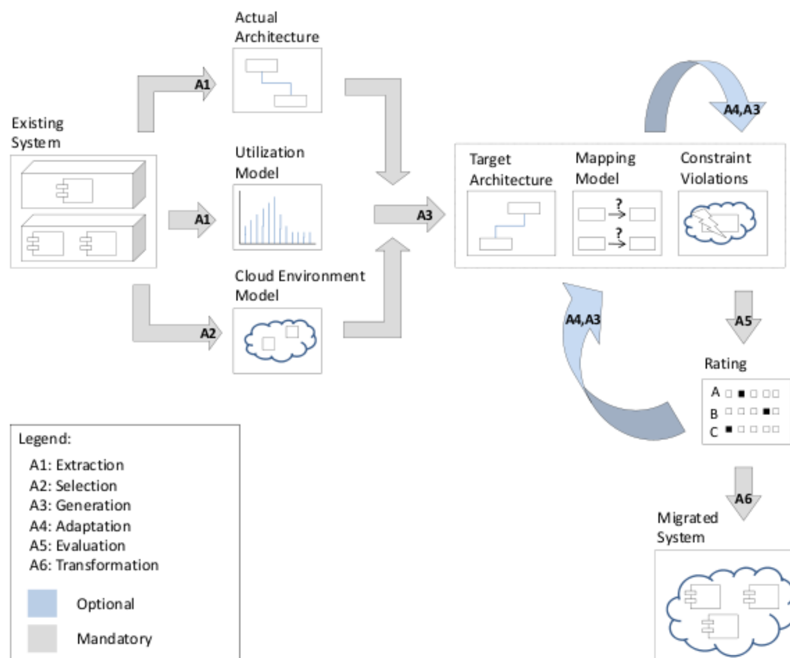

Figure 18: The CloudMIG approach. [18]

A2 - Selection: CloudMIG supports a variety of cloud environments and defines a meta-model to describe the common properties of those. To advance in the process of the migration, a specific instance of the meta-model has to be selected, describing the desired cloud environment.

A3 - Generation: This activity results in the generation of three artifacts: A target architecture, a mapping model and a constraint violation model, listing "the features of the target architecture which are non-conform with the cloud environment's specification." [18] The generation of this model is described in [19]. The Cloud environment constraints defined in the model for an environment are validated through a set of validators, that are plugged into the platform. Each validator thereby addresses a specific constraint; for example the *AbstractCommonTypeListManager* describes a handler for a predefined list of types, that create a violation, when used.

For the generation of the target architecture, CloudMIG defines three phases, containing the following:

P1 *Model transformation:* Features of the existing architecture are assigned to cloud-specific features.

P2 *Configuration:* This phase configures, how system feature resources are allocated, using rules and assertions for the heuristic computations in P3.

P3 *Resource-efficient feature allocation:* This phase improves the initial assignments from P1 regarding the resource efficiency using the rules and assertions of P2.

A4 - Adaption: Although the process automates much of the migration process, the generated results are not always satisfactory with respect to the requirements. Thus a manual adjustment has to take place, which is the purpose of this activity.

A5 - Evaluation: Before the last activity, an evaluation of the target architecture, generated by A3 and A4, takes place, involving static and dynamic analyses.

A6 - Transformation: At last, the legacy system is transformed manually according to the defined and generated models.

Activity A3 and A4 can be repeated either after an evaluation has been done, to improve the generated models, or after each generation process, analyzing the effects of different cloud environments on the legacy system.

3. Approach of this Thesis

> Science is the attempt to make the chaotic diversity of our sense-experience correspond to a logically uniform system of thought.
>
> – *Albert Einstein*

The *CloudMIG* approach and its implementation, *CloudMIG Xpress*, migrates existing legacy software systems to a cloud based environment. In this migration process a new architecture for the system is defined, that embraces the advantages of the environment, the "elasticity" in particular. To fit the legacy system onto this new architecture, it often has to be remodeled and restructured. In this process, the legacy system is converted with the help of the MoDisco framework to an instance of KDM (see Section 2.2.1 and 2.2.2, respectively). This transformation chain can be seen in Figure 19; MoDisco creates from a set of sources a Java model, which will then be transformed to a KDM instance. At the time of writing, MoDisco is only capable of transforming Java based legacy systems to KDM, therefore the transformation chain pictured contains Java source files.

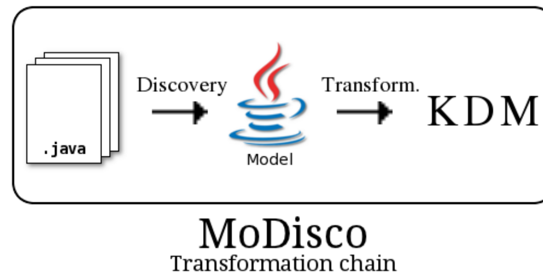

Figure 19: The transformation chain to a KDM instance in MoDisco.

3.1. Reverse Engineering of Java Class Files

As stated in Section 2.1.1 a problem arises, if the system, or parts thereof, is only available in binary or intermediate form; in the case of Java, a set of class files. For this case, MoDisco provides a mechanism to analyze the bytecode contained in these files, enriching the KDM instance with the result. However, this mechanism does not convert the class

file back to a Java source file, but merely extract signatures and members, ignoring the interesting part, the method bodies. In them, the vital parts of a class are contained, namely control and data flow information, that are necessary to enable the restructuring process. Thus, the provided mechanism is not satisfying.

The solution to this problem is a process, that is capable of extracting the whole information from a class file and the contained methods within. This process has then to be encapsulated in a *Discoverer*, that takes class files as input and returns KDM instances or entities and relations, respectively. Since projects consist probably of both class and source files, a better solution would be a uniform *Discoverer*, that can work with both types of files. Although an interesting approach, it would cross the boundaries of this thesis. Therefore another solution has to be found.

The second approach for a solution is defined in the first goal of this thesis (see Section 1.2.1) and introduces the use of a Decompiler. Because Java is based on bytecode, the decompilation process is much more simpler than in languages, that compile to native code (see Section 2.1.1). Thus, before the discovery of a KDM instance begins, the class files of the legacy system are put through a decompilation phase, leaving only Java source files to work with. This results in an extended transformation chain, depicted in Figure 20.

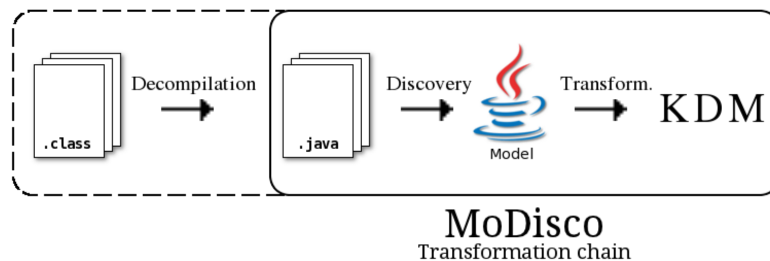

Figure 20: Extended transformation chain.

3.2. Dependency Analysis

For the migration process defined in the CloudMIG approach, the target cloud environment has to be selected. As stated in Section 2.2.3, CloudMIG Xpress provides a variety of cloud environments, including *Amazons EC2*[10] and the *Google App Engine*[11]. The essential parts are described through a meta-model, which is used in the transformation process of the legacy system.

An interesting part of the description is concerned with the constraint definitions, that are imposed on a system that runs on the selected environment. For example, The *Google*

[10] *http://aws.amazon.com/en/ec2/* (accessed 01.07.2012)
[11] *https://developers.google.com/appengine/* (accessed 01.07.2012)

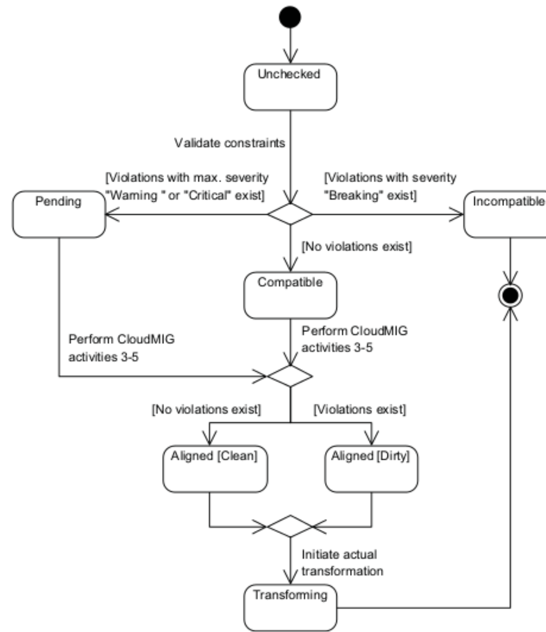

Figure 21: Possible states of a system in the constraint detection process. [19]

App Engine is very restrictive concerning system access. It disallows the use of threads, sockets, system calls and file system access, among others. Thus, when a legacy system is migrated to such an environment, it has to be validated against these constraints. Figure 21 shows the part of the workflow in CloudMIG Xpress related to constraints. Apparently, violations have a severity, that can break the migration process. But not all violations prevent the migration process (see [19] for examples).

Furthermore, if a legacy system is migrated, it often has to be redesigned in order to fit an architecture suitable for a cloud environment. In order to do so, the current system architecture has to be extracted and remodeled accordingly.

Both aspects addresses reachability: For the constraint violation examples given above, it is defined through inclusion and in the context of architecture remodeling, through the interconnection of the different system parts. As stated in Section 2.1.2, the transitive closure can be used to determine the reachability of system components. Because the system itself is described as an instance of KDM, the closure has to be computed on top of the entities and relationships defined therein, since KDM relationships are not transitive [36, page 66]. This reachability aspect and the implementation of it, respectively, is also the second goal of this thesis, defined in Section 1.2.2.

4. Java Decompiler

> If Java had true garbage collection, most programs would delete themselves upon execution.
>
> – *Robert Sewell*

The following sections describe existent Java decompilers, as well as the integration of one of them into CloudMIG Xpress. Section 4.1 gives an overview of the taken approach, while 4.2 describes Java decompilers. Section 4.3 defines test cases to evaluate the capacity of the decompilers. The results of the tests are analyzed in Section 4.4 and discussed in Section 4.5. Section 4.6 concludes with the integration approach into CloudMIG Xpress.

4.1. Overview

A stated in 3.1, the effort to create a new discoverer for the MoDisco tool crosses the boundaries of this thesis. The inherent problem of this approach is the decompilation of class files. This process is not trivial and is the core of entire theses [31; 10]. However, there exists a variety of Java decompilers, that can be used for the approach for a better bytecode support in CloudMIG XPress.

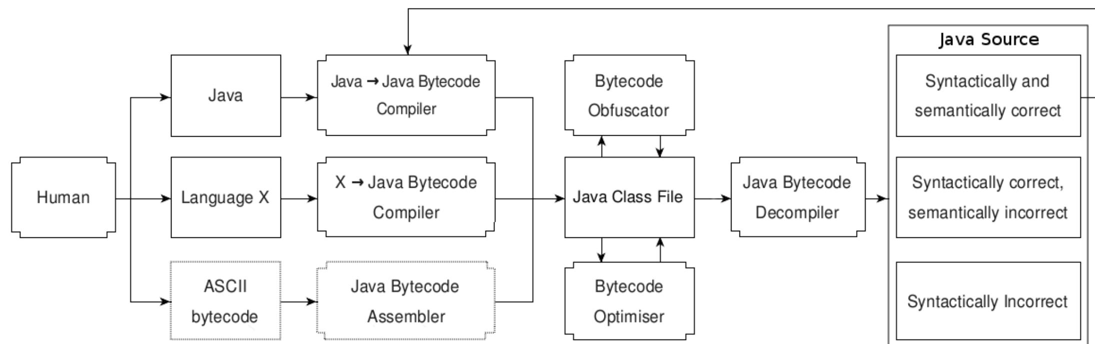

Figure 22: Compilation and decompilation workflow for Java bytecode [21].

A workflow of the compilation an decompilation process can be seen in Figure 22. Apparently, there is more than one way to create a Java Class file. For most Java

programs, *javac* is used. Others exists [15; 47; 42], but are not commonly applied. As of late, other languages make also use of the JVM [22; 37; 25; 34], delivering their own compiler; even the writing of a class file by hand is possible. All this leads to a syntactically correct class file, that can be optimized or obfuscated.[12] The use of such tools often change the structure of the file. It is still a functional equivalent to the original, but can lead to different semantics. Therein lies the problem of most decompilers: While they can often retrieve a syntactical correct Java source file, the generated code can differ from the original.

4.2. Available Java Decompilers

A variety of Java decompilers exists. This section gives a brief overview of most of them. The Decompilers mentioned here are selected from the first one hundred search results from the *Google Search Engine* using the terms "Java Decompiler" as well as from [45] and [21]. Those, that are not listed here, are either build onto the decompilers presented here, extending them with a GUI, or not available.

Mocha [46] Mocha was released as one of the first Java decompilers in 1996. The distribution also contained an obfuscator called *Crema*. Further development has stopped with the death of the author, but the decompiler is still available on several websites.

Jdec [6] Jdec is written in Java and an open source decompiler. It supports the decompilation of class files, that are compiled with *javac*. The latest version is 2.0 and has been released in May, 2008.

JODE [23] JODE is a package containing a decompiler and an optimizer. The current version is 1.1.2-pre1, released in February 2004.

Jad [26] Jad is a free for non-commercial use Java decompiler, that is no longer maintained. The last available version is 1.5.8.e/g, depending on the platform, and was released in 2001. Jad is an often used decompiler and is applied as the back-end of many other programs such as the *DJ Java Decompiler* [5].

Dava [32] Dava is a Java decompiler and part of the *Soot Java optimization framework* [28]. It aims at decompiling not only bytecode that is created through *javac*, but also arbitrary bytecode. Dava can be obtained through the Soot distribution, which is an ongoing research project. The current version is 2.5.0, released in January, 2012.

[12] A process to make the understanding of source or machine code harder on purpose.

JReversePro [27] JReversePro is written in Java and is a decompiler and disassembler. It is released under an open source license and currently in version 1.5.2, which was released in May, 2008.

Java Decompiler [40] This project aims at developing a platform independent decompiler, that also provides obfuscation options. The latest version is still in alpha phase and has been released in 2002.

Java Decompiler Project [13] This decompiler is a free for non-commercial use program aiming at decompiling Java 5 and higher byte code. The current available version is 0.3.3 and still in development.

4.3. Test Cases

As stated before, Java bytecode can be obtained through many different ways, including writing it down by hand. Each way is using his own strategy to create bytecode, which therefore can contain patterns, that are not easily translated back to Java structures. For example, a compiler developed for a language intended to run on the JVM, creates valid bytecode patterns, that don't correspond to a specific Java construct, much like instructions in the object code counterpart written in assembler. Thus, to fully test the capacity of a decompiler, all variants of how to assemble bytecode must be tested. This is a tedious and time consuming work and also crosses the boundaries of this thesis. Therefore, a much more simpler way of testing must be used.

The assumption of this thesis is a better support for the migration of *legacy software systems* in CloudMIG Xpress. To create tests in this context for a decompiler, the term has to be clarified. For this part of the thesis, a *legacy software system* is defined in the following way:

1. A software system written in Java, derived from the fact, that the MoDisco tool only supports Java source files.
2. The source files are compiled with *javac*. The compiler version has to be less or equal to 1.5.

The second definition results from the following observation: Java is an evolving language, so for each new major milestone, a new JDK is released. The current version of the JDK is 1.7 and was released in July, 2011. Programs written with this version are obviously not a legacy system in a strict sense. The most commonly used JDK version is still 1.6, released in 2006. Although systems compliant to this version could qualify for the term legacy software system, the language standard contained in this JDK does not differ significantly from the language standard defined in JDK version 1.5. Even though other compilers for the Java language exists, they are not commonly adapted, hence the above definition.

With this definition, a set of tests can be created, that cover the commonly used language constructs. After that, they are compiled with the different versions of *javac* to obtain a set of class files, that are then used with the different decompilers. The output of the decompilation process falls into one of three categories:

1. The decompiler creates no output.
2. The result is a syntactical correct Java program.
3. The result is syntactical correct and semantically equal to the original source code.

The following tests contain language constructs, that are present in most Java programs:

4.3.1. HelloWorld

The HelloWorld program is a very simple test, that contains everything needed to run a Java application. The program can be seen in Figure 6.

4.3.2. Inner Class

This test contains an inner class, that uses a member of the parent class. An excerpt containing the relevant code, can be seen in Figure 23.

```java
public class ClassTest {
    private String parent = "ClassTest";

    private class MyInnerClass {
        public MyInnerClass() {
            System.out.println("My parent is: " + parent);
        }
    }
}
```

Figure 23: Excerpt from ClassTest.

4.3.3. Conditions

This test contains the simple if-else construct, a switch statement as well as the C-Style if-else: `boolean_condition ? true_flow : false_flow;`

4.3.4. Exceptions

Exceptions are a frequently used construct in Java to delegate error handling. For this, a *try-catch* block is used. Even if a method does not throw an exception in the body, the method signature can force the use of a *try-catch* block. Also, to clean up the state, the *finally* keyword is introduced, whose body is always executed, if used after a *try-catch* block. The syntax of this block can be seen in Figure 24.

```
try {
    // attempt something
} catch(Exception ex) {
    // handle exception
} finally {
    // cleanup state
}
```

Figure 24: *Try-catch-finally* syntax in Java.

4.3.5. Loops

The two main loop types in Java, namely the *While* and the *For Loop* are tested here. The special case of the *For-Each Loop*, depicted in Figure 25, is not tested, because it is only so called "syntactical sugar", which will be translated into the normal *For Loop* construct in the compilation process.

This test is more interesting than it initially appears, because compilers, that translate a program to native code, transform loops often to *Do-While Loops*. One could assume, that this is also the case here.

```
for(Object object : objects) {
    // do something with object
}
```

Figure 25: *For-Each Loop* in Java.

4.3.6. Inheritance

Java is an Object-orientated language, so the testing of object inheritance is a matter of course. The two inheritance principles available in Java are the implementation of one or more interfaces and the extension of *one*[13] (abstract) class, respectively.

[13]Java does not support multiple inheritance of classes, but of interfaces.

4.3.7. Generics

Generics are a new Feature introduced with the Java 1.5 specification. It "allows a type or method to operate on objects of various types while providing compile-time type safety."[14] An example for *Generics* is shown in Figure 26. It displays the signature of the *ArrayList* class. The type *E* identifies the type of objects, that can be put into the *ArrayList*. Without *Generics*, the type has to be either *Object*, to enable the *ArrayList* for all kinds of objects, or there has to be an *ArrayList* for each object type that exists in the system and that is used in the context of an *ArrayList*.

```
class java.util.ArrayList<E>
    extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, Serializable
```

Figure 26: ArrayList signature with *Generics* in Java.

4.3.8. Annotations

Annotations are meta-data introduced in Java 1.5, that can be added to classes, methods, and variables. They can be obtained through the reflection mechanism in Java and are used to influence the runtime behavior. They can be identified through their notation, beginning with the @ symbol.

4.3.9. Enumerations

An enumeration type defines an order on related types. Normally, this would be done through a set of integers, where the name of the integer corresponds to the name of the type and its value to the position. The *Enum* class in Java encapsulates this behavior and enables the user to use an enumeration type like a class. Figure 27 shows an example of how days could be written as an enumeration type. Moreover, it shows, how the enumeration type can be used as class by defining a *String* representation for each day.

Other tests contained in the class files, that are not mention explicit, are the *Double Brace Initialization* pattern and the inline initialization of abstract classes (see Figure 28 and 29, respectively).

[14] *http://docs.oracle.com/javase/1.5.0/docs/guide/language/index.html* (accessed 01.07.2012)

```java
public enum Day {
    MONDAY("Monday"),
    TUESDAY("Tuesday"),
    WEDNESDAY("Wednesday"),
    THURSDAY("Thursday"),
    FRIDAY("Friday"),
    SATURDAY("Saturday"),
    SUNDAY("Sunday");

    private String stringRep;

    private Day(String stringRep) {
        this.stringRep = stringRep;
    }
}
```

Figure 27: Day enumeration type, as defined in the Enum test case.

```java
private static final Set VALID_CODES = new HashSet() {{
    add("XZ13s");
    add("AB21/X");
    add("YYLEX");
    add("AR2D");
}};
```

Figure 28: *Double Brace Initialization* pattern in Java.

```java
MyAbstractClass myAbstrctClass = new MyAbstractClass() {
    public void myAbstractMethod() {
        // implementation
    }
};
```

Figure 29: Inline initialization of an abstract class in Java.

4.4. Test Evaluation

The evaluation compares the original source files with the output of the decompilers. Because the files are rather small in code size, no special metrics are computed to show the degree of the difference; rather, they are compared manually.

The decompilers are rated by their output, which falls into one of the three categories stated in 4.3; a 0 is equivalent to the no-output category, while a 2 describes a syntactical and semantical correct output. Accordingly, 1 is syntactical correct. Additionally, two “in between” categories are introduced, referenced through 0-1, 1-2, due to the fact, that the output does not always correlate with one of the primary categories. For example, while the translation of a while loop into a for loop is a legal substitution, the semantics of both loops differ, thus the rating is 1-2 and not 2. Category 0-1 describes every output that is generated, but contains errors or stack dumps. The *try-catch-finally* result of most decompilers falls into this category, because the code either contains references to the stack or contains code, that does not equal the functionality of the original.

Unfortunately, the JDK in Version 1.1 and 1.2 will not run on modern systems, so the tests are only executed with the JDK versions 1.3, 1.4, and 1.5. However, it can be assumed, that the decompilers are also capable of translating files compiled with the former two versions of Java, because all but one decompilers had no problems translating the files compiled with the latter versions.

	HelloWorld	**Inner Class**	**Conditions**	**Exceptions**	**Inheritance**
Mocha	2	2	2	0-1	2, 2, 1-2
	Loops	**Annotations**	**Enum**	**Generics**	
Mocha	1-2	0	0	0	

Table 1: Test evaluation of Mocha.

Mocha The Mocha decompiler produces good test results. While some odd translations exists, like the *synchronized* in the class signature, the output is mostly semantically correct. The only problem occurs in the decompilation process of *try-catch-finally* constructs. Although the content of each statement of the construct is correctly translated, the keywords are replaced with a *pop*, indicating, that the stack representation at this point is used.

An interesting fact about the decompiler is, that it only accepts class files compiled with the compiler version 1.3; if attempted with higher versions, an error occurs. Thus, annotations, enumerations and generics could not be tested.

Jdec Jdec has problems with the translation of the C-style *if-else* syntax. While it translates the case to a standard *if-else* construct, the content of the statements are

	HelloWorld	**Inner Class**	**Conditions**	**Exceptions**	**Inheritance**
Jdec	2	2	1-2	0-1	2, 2, 2
	Loops	**Annotations**	**Enum**	**Generics**	
Jdec	1-2	2	1-2	1-2	

Table 2: Test evaluation of Jdec.

empty. It has also some quirks in the translation of a *try-catch-finally* construct. While the keywords exists and the content of the statements are also correct, a syntax error exists, in where an object is been referenced, before it is defined. Furthermore, the construct is wrapped in another *try* construct and two closing braces are missing.

The translations of the loops are rather unusual. The while-loop is indeed translated to a while loop, but the condition of the loop is put into an *if*-block in the body. The for loop does not exists, instead a while loop is used, where the condition and increment is also located in the body.

Most notably, Jdec is only one of two decompilers, that is capable of translating annotations. Not only are annotation definitions translated to the correct syntax, but it also discovers the use of annotations.

	HelloWorld	**Inner Class**	**Conditions**	**Exceptions**	**Inheritance**
JODE	2	2	2	2 (only 1.3)	2, 2, 2
	Loops	**Annotations**	**Enum**	**Generics**	
JODE	2	1-2	0	1-2	

Table 3: Test evaluation of JODE.

JODE The decompilation of most constructs are no problem for the JODE decompiler. It is capable of putting translated inner classes into the defining class file, rather than into a separate file; the double brace initialization pattern has also been correctly identified. The *try-catch-finally* statement has been successfully translated, but only if the file was compiled with the compiler version 1.3; 1.4 and 1.5 resulted in an error.

The new language features introduced with Java 1.5 have been recognized only partially. Annotations are translated to interfaces, while their use have not been discovered. Generics are only classes with the type *Object* isntead of T and enumerations could not be decompiled.

Jad Jad provides similar results as JODE. Exception handling is translated correctly, but also only in 1.3. But contrary to JODE, the decompilation of *try-catch-finally* does not end in an decompilation but a syntax error; instead of *try-catch-finally*, there is only

	HelloWorld	**Inner Class**	**Conditions**	**Exceptions**	**Inheritance**
Jad	2	2	2	2 (only 1.3)	2, 2, 2
	Loops	**Annotations**	**Enum**	**Generics**	
Jad	1-2	1-2	1-2	1-2	

Table 4: Test evaluation of Jad.

a " MISSING_BLOCK_LABEL_56" at the position of the keywords. While-loops are not recognized by Jad, instead a for loop with a missing iteration step is placed.

Also, as most other decompilers, the Java 1.5 language features are only partially translated; Annotations are interfaces, enumerations derive form the Enum class and the generic type T is replaced with *Object*.

	HelloWorld	**Inner Class**	**Conditions**	**Exceptions**	**Inheritance**
Dava	2	2	2	0-1	2, 2, 2
	Loops	**Annotations**	**Enum**	**Generics**	
Dava	2	1-2	1-2	1-2	

Table 5: Test evaluation of Dava.

Dava Although Dava aims at decompiling arbitrary bytecode and can therefore not rely on reversing the decompilation process of only one compiler, which has a predefined set of ways to create a class file, the output is useful in most cases. The only exceptions is, like in almost all decompilers that are tested here, the handling of *try-catch-finally*. It will be translated into a nested set of *try-catch* structures, in where the content of the *finally* statement is (partially) repeated.

Annotations and generics are also handled like in other decompilers. The enumeration is also a sub-class, but carries the keyword enum in its signature.

	HelloWorld	**Inner Class**	**Conditions**	**Exceptions**	**Inheritance**
JRP	2	2	0	0	2, 0-1, 0-1
	Loops	**Annotations**	**Enum**	**Generics**	
JRP	0	0-1	0	0	

Table 6: Test evaluation of JReversePro.

JReversePro This decompiler has problems with basic structures. Conditions, exceptions and loops have not been decompiled, while annotations are transformed to interfaces.

	HelloWorld	**Inner Class**	**Conditions**	**Exceptions**	**Inheritance**
JD	0-1	0-1	0-1	0-1	0-1, 0-1, 0-1
	Loops	**Annotations**	**Enum**	**Generics**	
JD	0-1	0-1	0-1	0-1	

Table 7: Test evaluation of Java Decompiler.

Java Decompiler The output of the Java Decompiler was mostly a set of errors. The output contained not much more than the class signature, if anything at all.

	HelloWorld	**Inner Class**	**Conditions**	**Exceptions**	**Inheritance**
JDP	2	0-1	2	2	2, 0-1, 2
	Loops	**Annotations**	**Enum**	**Generics**	
JDP	2	2	2	1-2	

Table 8: Test evaluation of Java Decompiler Project.

Java Decompiler Project The Java Decompiler Project is the best decompiler so far as measured by feature coverage. It decompiles every test with mostly equal semantics to the original except for two constructs. First, the double brace initialization pattern was not translated at all. Only the constructor call as well as the serial version uid could be recovered, while the content of the initialization, i.e., the addition of list elements, is lost. The second construct is the inline instantiation of classes. While the inline instantiation is semantically correct, the variable referencing the call of the constructor is replaced by "1 local1". The extended features of Java 1.5 are also correctly reversed, the only drawback is, that the generic attribute of classes is not used when instantiated.

4.5. Discussion

The overall test results can be seen in Table 9. The most notably deficit is the *try-catch-finally* construct. While a simple *try-catch* is translated by most of the decompilers, the *finally* statement seams the most hardest to translate.

The extended language-features introduced in Java 1.5 are also not recognized by most; annotations would simply be translated to interfaces while the use of the annotation is not discovered at all. Enumerations will be transformed to a sub-class of the *Enum* class and generics are simply ignored by translating the type T to *Object*.

Aside from these shortcomings, some interesting behavior can be observed:

- An inner class will be compiled into separate class file, referencing the parent class in the constructor. Therefore, most of the decompilers assume, that the inner class is a stand-alone class and generate a separate file.

- The direct initialization of member variables is replaced with constructor initialization.
- Missing constructors are added in the decompiling process.
- The explicit type of a variable is inferred. For example would an interface type be replaced with a class type implementing the interface.
- Some decompilers add the keyword *abstract* to the signature of an interface and his methods.
- Loops are not always translated back to the original representation, but to a similar with different syntax that achieves the same. For example the replacement of a for loop with a while loop, where the condition is put into the loop body.

	HelloWorld	**Inner Class**	**Conditions**	**Exceptions**	**Inheritance**
Mocha	2	2	2	0-1	2, 2, 1-2
Jdec	2	2	1-2	0-1	2, 2, 2
JODE	2	2	2	2 (only 1.3)	2, 2, 2
Jad	2	2	2	2 (only 1.3)	2, 2, 2
Dava	2	2	2	0-1	2, 2, 2
JRP	2	2	0	0	2, 0-1, 0-1
JD	0-1	0-1	0-1	0-1	0-1, 0-1, 0-1
JDP	2	0-1	2	2	2, 0-1, 2
	Loops	**Annotations**	**Enum**	**Generics**	
Mocha	1-2	0	0	0	
Jdec	1-2	2	1-2	1-2	
JODE	2	1-2	0	1-2	
Jad	1-2	1-2	1-2	1-2	
Dava	2	1-2	1-2	1-2	
JRP	0	0-1	0	0	
JD	0-1	0-1	0-1	0-1	
JDP	2	2	2	1-2	

Table 9: Test evaluation summary.

4.6. Integration into CloudMIG Xpress

After the evaluation of the test results, a suitable decompiler has to be chosen for the integration into CloudMIG Xpress. For this, the decompilers will be graded by the following criteria:

Language support Java is an evolving programming language and constantly adds new features to the language specification. With version 1.5 of this specification, Java introduced *Generics* and *Annotations*, among others, which are vital parts of some legacy systems. A suitable decompiler should support most, if not all, of those features.

Generated code quality The recreation of the source-code form the bytecode is not perfect, for example names and comments cannot be recovered,[15] but the produced output should be approximately equal to the original concerning the semantics. For example: *Generics* should not be translated to *Object* or the *Enum* type not be changed to a class.

Compatibility The decompiler must work fully automatic and with no user guidance. If configuration has to be done, this has to happen before the decompilation process. Also, a No-GUI mode is mandatory.

Support As stated above, Java is evolving and so should the decompiler. An ongoing development is preferable.

License Because the decompiler will be integrated into CloudMIG Xpress, the license of the decompiler should not interfere with this aim. Thus, an open source decompiler is mandatory, but not sufficient, because even an open source license (like the GPL) could stand in the way of the integration.

Decompiler	**latest release date**	**License**
Mocha	1996	freely distributable
Jdec	May, 2008	GPL
JODE	2004	GPL/LGPL
Jad	2001	Free for non-commercial use
Dava	January, 2012	LGPL
JReversePro	May, 2008	GPL
Java Decompiler	2002	Public Domain
Java Decompiler Project	August, 2010	Free for non-commercial use

Table 10: Decompilers support status and their license.

An overview for the last two criteria can be found in Table 10. Because the license is the exclusion criterion, only Mocha, JODE, Dava and the Java Decompiler are candidates

[15] At least not, when the code is not compiled with the debug option.

for an implementation. The language support narrows the selection down to Dava and JODE, with equal support values.

For the construction of the KDM instance and the subsequent analysis, information is a valuable asset, thus Dava will be selected for the implementation, because in contrast to JODE, it is capable of decompiling enumerations. Also, it is the only decompiler that has an ongoing support.

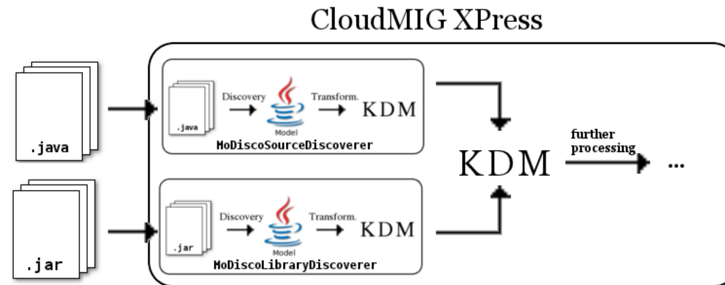

Figure 30: System to KDM transformation process in CloudMIG Xpress.

Figure 30 shows the current approach of CloudMIG Xpress on how to create an instance of KDM from the Java legacy system. The system is separated into Java source files and JARs, representing the used libraries of the legacy system. For both file-types a discoverer is implemented in MoDisco, each creating a part of the overall instance of KDM.

Because the JARs are the only source for class files, the Dava decompiler will be hooked into this approach as follows:

1. Before the overall discovery process starts, the JARs will be extracted.

2. Each generated folder will be searched for class files to collect them.

3. All collected class files from one folder will be decompiled by Dava.

4. The output of the decompilation process will be a set of Java source files, that correspond to a set of class files. Each class file in this set will be replaced with a source file.

5. The MoDisco source discoverer takes the modified folders and the original source files as input and generates a KDM instance.

With this approach, the library discoverer becomes obsolete. The downside is, that the differentiation between libraries and sources vanishes, making it harder to argue in the restructuring process.

5. KDM-based Dependency Analysis

> Nothing is so good that somebody, somewhere will not hate it.
>
> – *Pohls Law*

This section describes the implementation of a dependency analysis based on KDM into CloudMIG Xpress. Section 5.1 gives a brief overview, Section 5.2 describes the taken approach, while Section 5.3 shows the implementation of the approach.

5.1. Overview

The interconnections of system components are a crucial information in addressing the restructuring of the system for migration. As shown in Section 3.2, this information can be used to determine the severity of constraint violation regarding the use of prohibited classes, for instance. The discovery of such interconnections can be found through a dependency analysis by traversing the relationships of each system component.

CloudMIG Xpress uses instances of KDM to describe legacy software systems in a uniform way, so a dependency graph is inherently given. This derives from the fact, that for each entity the set of relationships to other entities are discovered in the construction of the KDM instance. Some of those relationships, in particular those that are contained in the Code and Actions package, like *Imports*, *Implements* or *Calls*, are mappings of dependencies.

But as stated also, knowing the direct dependencies is not always enough if used in the context of reachability. Therefore, the transitive closure has to be computed, which is the content of the next section.

5.2. Approach

To compute the transitive closure, the dependency graph is needed. The graph is represented through instances of (sub-classes) of KDMEntity and KDMRelationship, respectively. But not all entities and relationships are needed, because they represent irrelevant information for the closure, for example, the relation between a KDM *MethodUnit* and the KDM *BlockUnit* is irrelevant, because the latter is a grouping element of more interesting KDM *ActionElements* describing control and data flow capabilities. So before an algorithm can be defined, the following limitations have to be introduced:

- For the reachability question, only structure descriptions in the Code domain are relevant, so only entities and relationships from the Core and Actions package will be considered for the closure computation.
- Since the primary concerns are Java legacy systems, the closure will be computed only on Java relevant entities and relationships. Those are:
  - *ClassUnit* to describing a Java Class.
  - *InterfaceUnit* to describe a Java Interface.
  - *MethodUnit* describes a method within a Java Class.
  - *Signature* of a *MethodUnit*.
  - *ParameterUnit* as part of a *Signature*.
  - *StorableUnit* for local and member variables.
  - *Implements* describes the implementation of a Java Interface.
  - *Extends* describes the extends relationship of a Java Class.
  - *Imports* describes the use of classes and interfaces from other packages.
  - *Calls* to describe a calling relationship between two methods.

With these limitation in mind, the algorithm can be defined. Figure 31 shows a demonstration of the discovery algorithm for the transitive closure, which works as follows:

1. For each entity encountered, that has not been visited yet, a node will be created and a depth-first search on the entity's relationships is started.
2. For each node, that is encountered on the traversal and that has not been visited yet, a node will be created. If the traversal cannot go further in the dependency graph, that is, if a node either has already been visited or has no dependencies, an edge between the current visited node and its predecessor is created. (See step 4. and 5.)
3. If a node has outgoing edges and an edge between the node and its predecessor has been created, a transitive edge is created from the predecessor to each node that is reachable through these edges (see 9. - 11.).
4. The traversal stops, if all reachable nodes from the start node have been visited.

If this algorithm is executed on the example given in Figure 32, the result of step 3 would be two transitive edges (labeled as a and b), which are *indistinguishable*.

One could ask, if they have to be distinguishable. If they stay indistinguishable, the only information, that can be derived is, that the node labeled 1 can *somehow* reach the node labeled 4. If they could be distinguished from another, the derived information

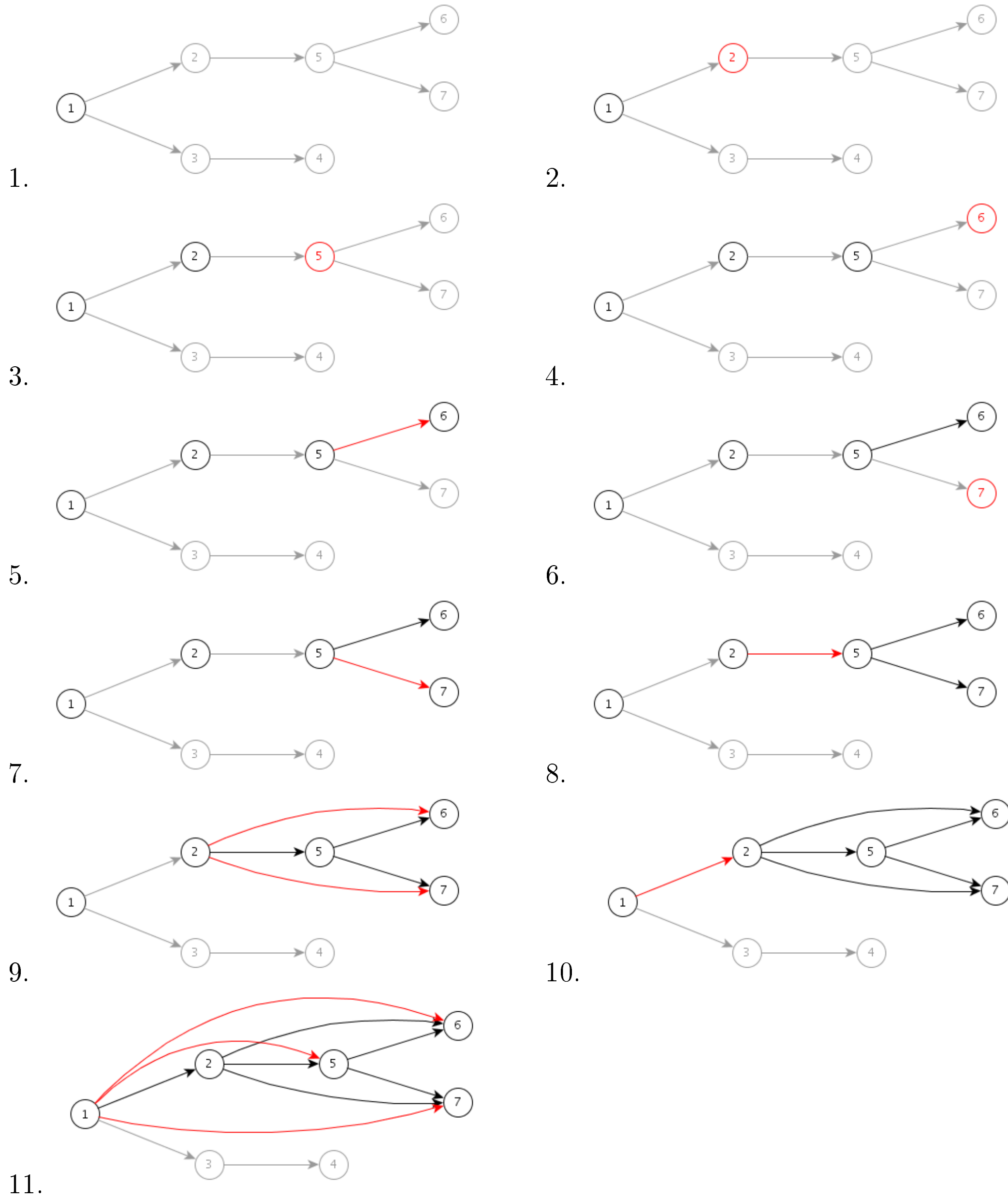

Figure 31: Discovery algorithm example for the transitive closure.

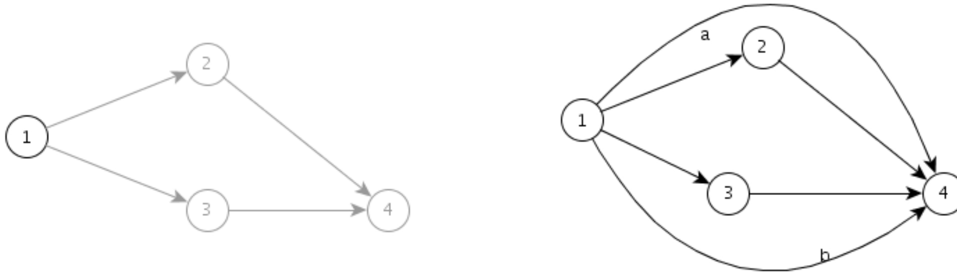

Figure 32: Double transitive edge.

would be, that node 1 can reach node 4 on two different paths. So, the first case loses valuable information. Step 3 has thus to be modified to contain a distinction mechanism for transitive edges. To gain as much information of such an edge, the edge will be labeled with the path, that leads from the start-node to the end-node. For the example given, a and b would be labeled ((1,2),(2,4)) and ((1,3),(3,4)), thus giving the advantage of not only knowing, that node 1 can reach node 4 in two different ways, but also on which path node 1 can reach node 4.

5.3. Implementation

The implementation should consider the two following aspects:

1. The approach has to be easily extendable. Even if the focus of this thesis evolves around Java, others may use the same approach to discover a transitive closure for legacy systems written in C, where the main components are functions and primitive data structures.

2. Each node should contain as much information as possible, thus each node does not only contain outbound edges, but also inbound edges. This is due to the fact, that not only reachable aspect from one system part is considered, but also the reversed aspect, i.e., which system parts are depend on a the currently viewed part. Also, considering the constraint violation aspect, each node should contain the reference to the code area, where it is defined. Therefore, the KDMEntity considered with the node, will be wrapped. The same is true for the edges of the graph, which therefore also wrap the KDMRelationship, that constitute the dependency.

5.3.1. Nodes of the Closure Graph

Figure 33 shows the class diagram of the *AbstractNode* class used for the transitive closure graph. It contains the wrapped CodeItem and also the lists of incoming and outgoing

relationships. Furthermore, it contains the the qualified name of the wrapped CodeItem, like *java.lang.File*, as well as the type of the wrapped CodeItem, for example a Class.

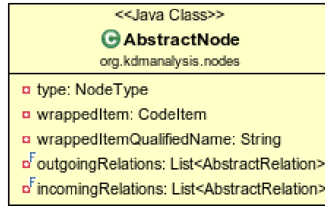

Figure 33: Class diagram of *AbstractNode*.

Derived thereof are three concrete node types, representing those components of the system, that are of interest for the analysis, namely classes, interfaces, and methods. The class diagram given in Figure 34 shows the node classes that represents these components.

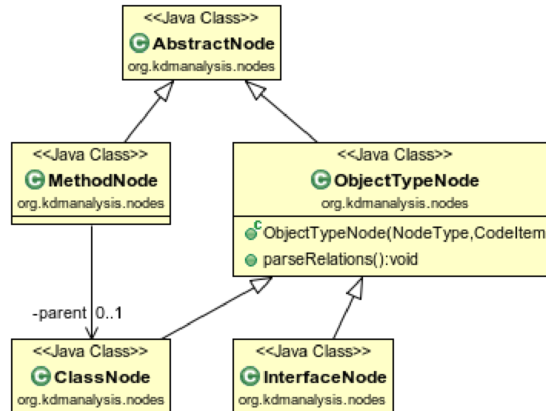

Figure 34: Concrete nodes for the transitive closure graph.

The *ObjectTypeNode* is the abstract parent of *ClassNode* and *InterfaceNode* and contains methods to parse the contained relationships in a *ClassUnit* and *InterfaceUnit*.

MethodNode is a special type of node, which only makes sense in the context of a *ClassNode*. While a *MethodNode* is contained in the closure graph, all its relations are also contained in the corresponding parent *ClassNode*, rendering it in most use cases obsolete. Furthermore, a *MethodNode* contains only instances of the *CallsRelation* class (s.b.), limiting its usefulness even more. However, there are two use cases where a *MethodNode* can be useful:

"Prohibited" classes While mostly whole classes are blacklisted, cases exists, that only prohibit the use of certain methods of a class. The lookup of such methods could

in fact be handled through the lookup of the parent *ClassNode*, traversing the relationships of the node in search for a *CallsRelation*, that has the to-field set to a *MethodNode* wrapping the *MethodUnit*, but this process has to be done manually, while a *MethodNode* automates this process.

Control flow Sometimes it is useful to determine the control flow in the system from a certain point. In such a case the interesting part is the calling hierarchy of methods, that has to be, in the absence of a *MethodNode*, manually be crafted by accessing the relationships of the *ClassNode* that declares ownership of the method.

5.3.2. Edges of the Closure Graph

Figure 35 contains a class diagram for the relationships, that can occur in the closure graph. *AbstractRelation* is the abstract parent class of all relationships and contains the *KDMRelationship* that forms the relationship, as well as the nodes wrapping those entities, that constitute the relationship.

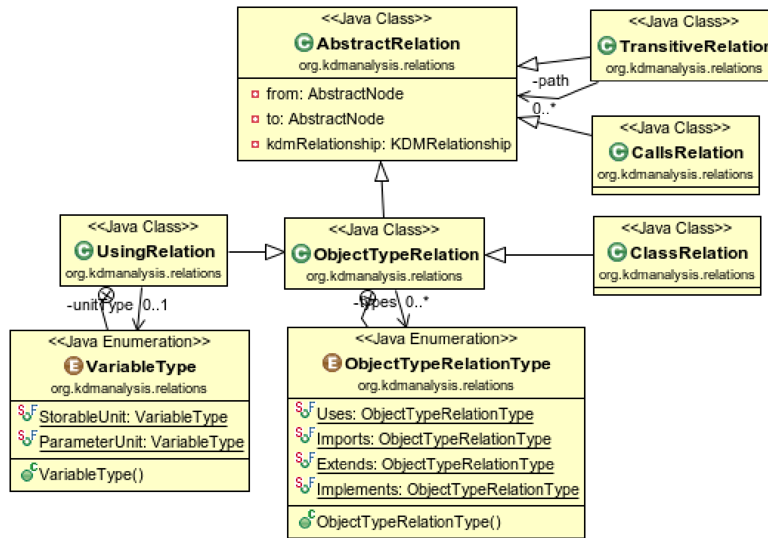

Figure 35: Relationship-classes for the closure graph.

CallsRelation This relation type wraps the corresponding KDM *Calls*-Relationship. Furthermore, it constrains the type of nodes that can be referred through the *from* and *to* members of *MethodNodes*. Like *Calls*, this relation type is used to describe the control flow from one method to another.

ObjectTypeRelation This relation type is used to describe the fact that a relationship between two classes or interfaces exists, respectively. Instances of Sub-classes of this

relation type specify the semantic of the relation. Furthermore, this class defines the enumeration type *ObjectTypeRelationType* describing the nature of the relation. While *Imports*, *Extends* and *Implements* should be obvious, the *Uses* needs some further explanation. *Uses* describes the fact, that a Class uses another Class or Interface as a member type or in a method and can be found in the context of the *UsingRelation* (s.b.). Also, if a class or interface extends or implements another class while an import relationship is present, then the two relationships are fused together, defining a relationship that has two types, namely (*Imports*, *Extends*) or (*Imports*, *Implements*).

ClassRelation An instance of a ClassRelation is more or less "syntactic sugar", because it adds nothing to the *ObjectTypeRelation*. Its sole purpose is to describe a relation that can have the two types defined in the *ObjectTypeRelation* and to separate such relationships from a *UsingRelation*.

UsingRelation To describe the fact, that a class "uses" another class as either a member or in the context of a method, as a parameter or local variable, the *UsingRelation* is used. It also defines an enumeration type, describing the class of the wrapped *KDMEntity*, which is either a *StorableUnit* or a *ParameterUnit*.

TransitiveRelation Perhaps the most important relation, the *TransitiveRelation* is used to describe transitive edges in the closure graph. For the distinguish-constraint defined in 5.2, the relation contains a variable named *path*, containing all relationships on the path from the start-node to the end-node in a *LinkedList*.

5.3.3. Traversing a KDM Instance

To avoid the parsing of the XMI file, that contains the KDM instance of the legacy system in XML manually, EMF is used. It transforms the XMI to a set of classes mapping the entities and relationships contained within, to allow a more easy way of traversing the model graph. With this premise and the requirement, that the whole project has to be a Java library, the analysis class shown in Figure 36 can be defined.

The two methods given in the interface are used to parse one or more instances of KDM[16], determined through first parameter; the class *Resource* thereby contains a KDM instance. The second parameter, the *LookupType*-Array, defines the set of nodes to look-up. A LookupType wraps the qualified name of the entity to look for, as well as the type of the entity. For example, to search for Java's *File*-Class, one would create a *LookupType* containing the name "java.io.File" and the *ClassUnit*-Class.

The return value of the methods is a list containing all node-instances, that are present in the closure graph and match a LookupType. Reconsidering the example with Java's *File*-Class, the result would be a list, containing only one node maximum, which wraps

[16]whereby the first method, *parseKdms*, relies internally on the second method, *parseKdm*.

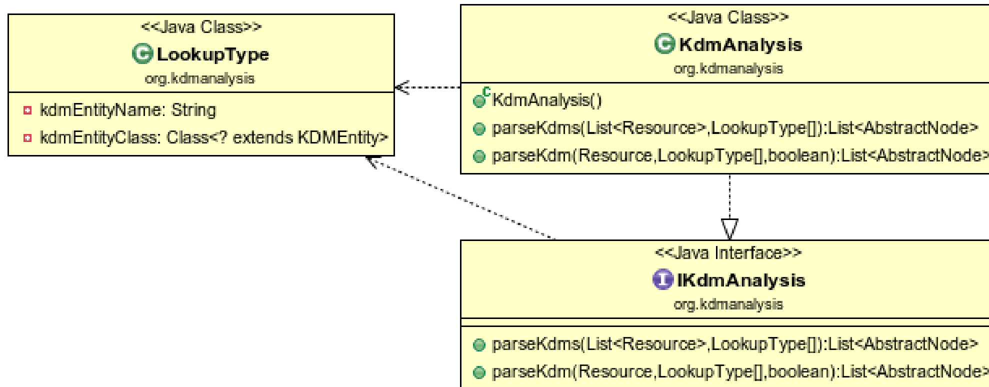

Figure 36: Interface of the dependency analysis library.

the *ClassUnit* describing the *File*-Class, if, and only if, the class is used somewhere in the system.

The boolean parameter in the second method is used in the context of parsing only one instance of KDM at the time. It preserves the result graph of previous parse-calls, if set to *true*.

The actual parsing or traversing of the model works as follows:

1. From the given resource, the *Segments* (see "the package named KDM" in section 2.2.1) are extracted.

2. All *CodeModel* instances contained in a *Segment* are extracted.

3. For each *AbstractCodeElement*, which is a direct child[17] of the *CodeModel*, a *Handler*-object is called to delegate the traversal of that entity.

The class diagram of all currently available handlers is shown in Figure 37. The purpose of these handlers is the creation of nodes and edges for the graph, i.e., creating instances of the above defined nodes and relations. Since the base of this thesis is the Java programming language, the handlers defined here are mostly concerned with handling entities representing constructs of this language. For other programming languages, more handlers have to be defined, which is a rather easy task, with the approach presented here; for each construct of interest in the language, a handler class extending the *AbstractEntityHandler* has to be created.

[17]Direct children of *CodeModels* are all *AbstractCodeElements* that are contained in the *CodeElement*-List of a *CodeModel*.

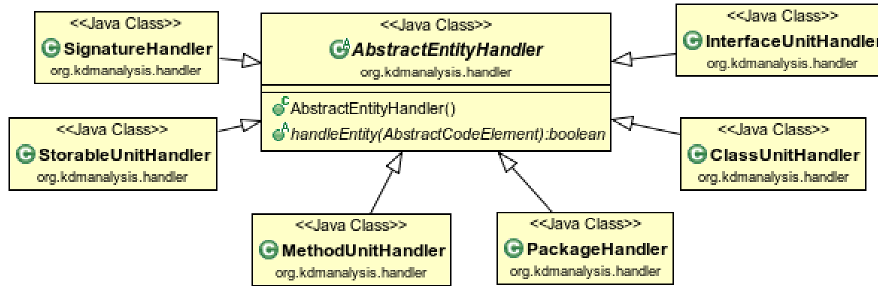

Figure 37: Entity-Handlers that are currently defined in the analysis library.

The further traversal of the graph will be achieved through delegation — a handler can call other handlers. For example, the *ClassUnitHandler* will call the *MethodUnitHandler* for each method a *ClassUnit* contains.

The last part of the implementation addresses the visited state. When a relevant KDM entity has been visited, a node is created. But the entity itself cannot be manipulated to save a reference to the node and thus cannot be marked as visited. To solve this problem, a *NodeManager* is introduced; the class diagram for it can be seen in Figure 38.

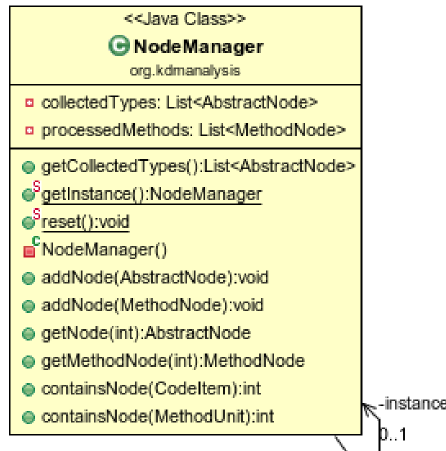

Figure 38: The *NodeManager* class diagram.

The *NodeManager* collects all nodes that are created in the *collectedTypes* and *processedMethods* list, respectively. It is implemented as a Singleton and contains methods to lookup and retrieve nodes.

The reason behind the distinction of *MethodNodes* from all other nodes is performance: If a lookup for a method is made, the algorithm needs only to search in a subset of nodes.

6. Evaluation

> Evaluate what you want — because what gets measured, gets produced.
>
> – *James Belasco*

The following sections contain the evaluation results for the decompiler, as well as the dependency analysis library. Section 6.1 gives an overview of the evaluation, while Section 6.2 contains the test results for the Dava decompiler with real software systems. The section concludes with the evaluation of the KDM-based dependency analysis in Section 6.3.

6.1. Overview

This thesis is based on the two main concepts of a decompiler integration into CloudMIG Xpress and the creation of a dependency analysis library for instances of KDM. While the sections 4 and 5 describe the approach and implementation details for these concepts, an evaluation of the outcome has yet to be done.

In order to create a conclusive result for the decompiler, the evaluation of it is done with a set of real live systems:

MyBatis JPetStore is an application based on suns original J2EE Pet Store, which was designed as a showcase for development of AJAX-enabled web applications.

JForum is a discussion board implemented in Java. The current stable release is 2.1.9.

For the dependency analysis on KDM instances, test systems have to be created, that cover all cases, in where a constraint violation can occur. Also, the reachability of those violations will be addressed.

6.2. Evaluation of the Dava Decompiler

To test the decompiler, the libraries for the test-applications are decompiled and random samples compared to the original source file with regard to the tests defined in Section 4.3. Also, some performance and output metrics are computed to show the overall efficiency.

The platform, on which the decompilation process has run, is specified as follows:

- Intel Core i5 2500K 4×3.30GHz
- 2×4096GB DDR3-1600 Ram
- Windows 7 Professional

The metrics are computed with the help of the Windows Task Manager and the timer included in the Soot project.

6.2.1. MyBatis JPetStore

Overall decompilation time	6h
Used RAM in decompilation process	Lowest: 1 GB/ Highest: 1,9 GB
Overall .class files	4023
Decompiled .class files	3805
Output ratio	94,6%

Table 11: JPetStore Performance metrics.

All but one JAR could be decompiled with Dava. Unfortunately, the decompilation process has to be stopped two times to remove classes form the hsqldb library that created an infinity loop.

6.2.2. JForum

Overall decompilation time	8.5 h
Used RAM in decompilation process	Lowest: 800 MB/ Highest: 1,3 GB
Overall .class files	7050
Decompiled .class files	3615
Output ratio	51,3%

Table 12: JForum Performance metrics.

In this test case, more than one JAR could not be decompiled with Dava. Also, like in the JPetStore test case, the decompilation process has to be restarted, because of faulty behavior of the decompiler.

6.2.3. Output Evaluation

Inner classes Inner classes as well as the inline instantiation are, as stated, compiled to separate files named *ParentClass$InnerClass* and *ParentClass$1*, respectively. Dava handles both class files as separate classes, so the output generates more than one Java source file. This is also the case in all probes examined, where inner classes are mostly used to encapsulate local behavior.

Conditions An often encountered construct in methods is the exit condition at the beginning of the method body, to stop further execution of the method, if requirements are not hold. The generated output turns this behavior over, executing code only, if the requirements are hold, as can be seen in Figure 39.

An also interesting case is the transformation of *if-else if* constructs. They are converted to nested *if-else*s with the condition reversed. For example, the common use case of *if-else if* is a emulation of the *switch*, where conditions successively tested for *true*. The generated output is reversing those conditions, testing for *not true* and nesting the *else if* statements as *if* statements in the body of the first *if*.

org.apache.log4j.AppenderSkeleton

```
if(!isAsSevereAsThreshold(event.getLevel())) { return; }
// method code
```

Dava output

```
if (this.isAsSevereAsThreshold(r1.getLevel())) {
    // method code
}
```

Figure 39: Exit condition example.

Loops Loops are often not translated back to the original counterpart, but to while or for loops, as can be seen in Figure 41. However, an interesting part is how code used around the loop is handled. Instead of positioning it before the the loop statement, the code gets initialized in the initialization part of the loop. An example for this behavior can be seen in Figure 40. Also, some loops are replaced with what seems to be an optimized counterpart. An example for this can be seen in Figure 42.

org.objectweb.asm.ClassWriter

```
interfaceCount = interfaces.length;
this.interfaces = new int[interfaceCount];
for (int i = 0; i < interfaceCount; ++i) {
    this.interfaces[i] = newClass(interfaces[i]);
}
```

Dava output

```
for (n = r4.length, o = new int[n], i2 = 0; i2 < n; i2++) {
    o[i2] = this.newClass(r4[i2]);
}
```

Figure 40: Loop example in JPetStore.

Exception handling Most *try-catch* statements are decompiled correctly, however, sometimes they are missing completely, as depicted in Figure 41. But this behavior occurs arbitrary and do not seem to correspond to the number of *catch* statements; for example, the class *BeanUtils* in *org.springframework.beans* contains the method *instantiateClass* with four *catch* statements, decompiled correctly.

org.springframework.beans.AbstractPropertyAccessor

```
for (PropertyValue pv : propertyValues) {
    try { setPropertyValue(pv); }
    catch (NotWritablePropertyException ex) {
        if (!ignoreUnknown) { throw ex; }
    }
    catch (NullValueInNestedPathException ex) {
        if (!ignoreUnknown) { throw ex; }
    }
    catch (PropertyAccessException ex) {
                if (propertyAccessExceptions == null) {
                        propertyAccessExceptions =
                            new LinkedList<PropertyAccessException>();
              }
        propertyAccessExceptions.add(ex);
    }
}
```

Dava output

```
r4 = r8.iterator();
while (r4.hasNext()) {
    r5 = (PropertyValue) r4.next();
    this.setPropertyValue(r5);
}
```

Figure 41: Missing *try-catch* statements.

Inheritance Inheritance has been always recognized correctly in the probes. Interfaces and their methods contain the *abstract* keyword, but are otherwise correctly decompiled.

net.sourceforge.stripes.controller.DispatcherHelper

```
Class<?> temp = type;
while ( temp != null ) {
    for (Method method : temp.getDeclaredMethods()) {
        Class[] args = method.getParameterTypes();
        if ((method.getAnnotation(ValidationMethod.class) != null) &&
            ((args.length == 0) ||
             (args.length == 1 && args[0].equals(ValidationErrors.class)))) {
                      validationMethods.add(method);
        }
    }
    temp = temp.getSuperclass();
}
```

Dava output

```
r5 = r0;
while (r5 != null) {
    r6 = r5.getDeclaredMethods();
    i0 = r6.length;

    if (0 >= i0) {
        r5 = r5.getSuperclass();
    } else {
        r7 = r6[0];
        r8 = r7.getParameterTypes();

        label_5:
        if (r7.getAnnotation(class
            "net/sourceforge/stripes/validation/ValidationMethod") != null) {
            if (r8.length != 0 &&
                (r8.length != 1 ||
                !(r8[0].equals(class
                 "net/sourceforge/stripes/validation/ValidationErrors")))) {
                break label_5;
            }
            r4.add(r7);
        }
    }
}
```

Figure 42: Replaced For-Loop.

org.springframework.beans.AbstractPropertyAccessor

```
package org.springframework.beans;

public abstract class AbstractPropertyAccessor
    extends PropertyEditorRegistrySupport
    implements org.springframework.beans.ConfigurablePropertyAccessor
```

Figure 43: Class signature with interface implementation.

However, an oddity sometimes happens, that an interface, that is packed in the same package as an implementing class, is prefixed with the qualified path. An example for this can be seen in Figure 43.

Generics The generic type declaration in class and interface signatures are omitted in the decompilation process, as depicted in Figure 44. However, instead of translating the generic type *T* to *Object* in this example, permitting all types in the context of the class, it is translated to *Comparable*, from which *T* derives.

net.sourceforge.stripes.util.Range

```
public class Range<T extends Comparable<T>>
        implements Comparable<Range<T>> {
    private T start, end;
    // ...
```

Dava output

```
public class Range implements java.lang.Comparable {
    private Comparable start;
    private Comparable end;
    // ...
```

Figure 44: Generics and the decompilation counterpart.

Annotations As stated in the test results for Dava, annotations are not fully understand, as can be seen in the example provided in Figure 45. However, annotating classes, methods, and variables is only one use case of annotations; another is the retrieving. This retrieving is done through the method *getAnnotation* of the Class type. To use this method, the class of the annotation, that should be retrieved, must be passed as the parameter. The decompilation output of such attempt can be seen in Figure 42: The *class* keyword is used along with the qualified name of the class to retrieve.

Enumerations An example for enumerations is given in Figure 46. An enumeration is translated to a class-like structure, constructed with the help of integers.

org.apache.ibatis.annotations.Arg

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Arg {
    // ...
```

Dava output

```
public abstract annotation interface Arg
    extends java.lang.annotation.Annotation
```

Figure 45: Annotations in Dava

6.2.4. Discussion of the Evaluation

Compared to the original source files, the output of the decompiler is in most cases equivalent; exceptions are shown above. However, it is not always clear, if the difference between the original and the output is caused by the decompiler or the used compiler. For example, *javac* extracts inner classes into separate files, which are interpreted by the decompiler as separate classes.

The usefulness of the output of the decompiler is the last question, that has to be answered. As stated and shown in the given examples, names of variables and comments cannot be retrieved, so the meaning of a decompiled file is harder to derive. But in the context of this thesis, this case is negligible; the interesting part is the resembled code structure. Under this aspect, the output is useful in most cases, because control and data flow is often represented correctly.

6.3. Evaluation of the KDM-based Dependency Analysis

For the evaluation of the KDM-based dependency analysis, tests have to be formulated that cover all usage contexts of classes or interfaces. These can be divided into indirect and direct usage, depicted in Figure 2 and 47, respectively. Direct usage covers the following constructs in a class or interface:

Imports If class A uses or derives class B's functionality, B has to be made visible in the execution context of A. For this, B will often be *imported* into A through the path describing the location of B.

Extends/Implements A class can extend another class or implement the functionality of an interface.

net.sourceforge.stripes.controller.LifecycleStage

```
public enum LifecycleStage {
    ActionBeanResolution, HandlerResolution, BindingAndValidation,
    CustomValidation, EventHandling, ResolutionExecution,
    RequestInit, RequestComplete }
```

Dava output

```
public final enum class LifecycleStage extends Enum {
    public static final enum LifecycleStage ActionBeanResolution;
    public static final enum LifecycleStage HandlerResolution;
    // ...
    private static final LifecycleStage[] $VALUES;

    public static final LifecycleStage[] values() {
        return (LifecycleStage[]) $VALUES.clone();
    }

    public static LifecycleStage valueOf(String r0) {
        return (LifecycleStage) Enum.valueOf(class
            "net/sourceforge/stripes/controller/LifecycleStage", r0);
    }

    private LifecycleStage(String r1, int i0) { super(r1, i0); }

    static {
        ActionBeanResolution = new LifecycleStage("ActionBeanResolution", 0);
        HandlerResolution = new LifecycleStage("HandlerResolution", 1);
        // ...
        LifecycleStage[] $r8 = {
            ActionBeanResolution, HandlerResolution, //...
        };
        $VALUES = $r8; }}
```

Figure 46: An enum type compared to the decompiler output.

Member To hold states in a class, member variables are used.

Signature The input and output for a method is defined here.

Local variables To compute output, sometimes a temporary state has to be hold, that is irrelevant to the overall state of the class. For this, local variables are used.

```
import com.prohibited.DoNotUse;

public class CantBeMigrated
    extends DoNotUseEither
    implements ProhibitedInterface {

    private DoNotUse member;

    public ProhibitedClass myMethod(IAmAlsoNotAllowed instance) {
        ProhibitedClass local = new ProhibitedClass(instance);
        return local;
    }
}
```

Figure 47: Places in code, where prohibited classes can occur.

The indirect usage of a class is defined over a direct usage of an *intermediate* class. For example, if class A imports class B, that imports class C, then A would use C indirectly, through the intermediate B.

6.3.1. Evaluation Tests

In order to determine the detection effectiveness for the use cases described before, the following tests are defined, containing one or more classes, that cover one or more of the use cases. The test cases are implemented as *JUnit*[18] tests and executed in a *Maven*[19] build process, therefore the tests are defined as a set of assertions.

Inheritance This simple test defines the interface *ITestInterface*, that is implemented by the class *TestImpl*. The corresponding UML class diagram can be seen Figure 48. The task of the test is to detect the *imports/implements* relationship between them.

MethodAndMemberVar This test describes the use of a class or interface as either a member of the class or as a local variable in a method. Figure 49 shows the class used

[18]http://www.junit.org/ (accessed 01.07.2012)
[19]http://maven.apache.org/ (accessed 01.07.2012)

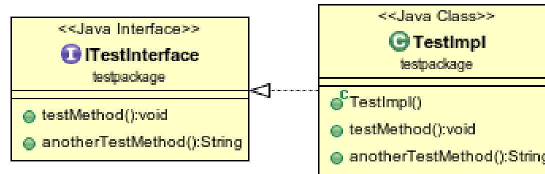

Figure 48: InheritanceTest class diagram.

for this purpose. It implements a member called *memberVar* of type *java.lang.System* as well as a method *localVar* containing a variable referencing the *java.io.File* class.

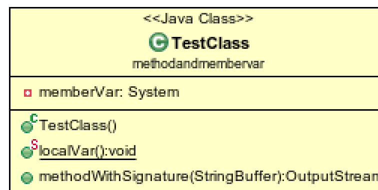

Figure 49: Test class containing a member and a local variable.

Signature This test refers also to the class depicted in Figure 49. The test describes the use of classes in a method signature; in this example, the use of *java.lang.StringBuffer* as a parameter and *java.io.OutputStream* as a return value.

Tricky This test is a special test, because it uses the *java.io.File* class as follows:

```
java.io.File.createTempFile("test", "txt");
```

This use case is neither covered by an import statement, nor is it assigned to a local variable and therefore harder to discover. Nevertheless, this use case must also be detected.

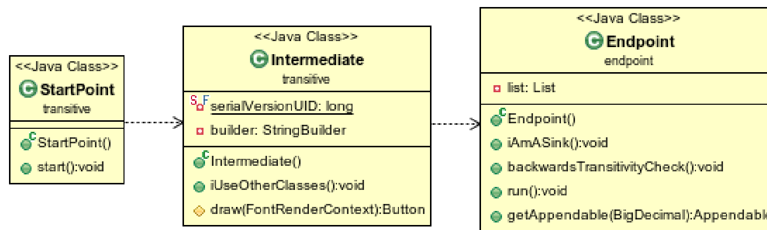

Figure 50: Test classes for transitivity tests.

Transitive To test the detection of classes outside of the system, i.e. in layer 2 and 3, the test classes shown in Figure 50 are defined. *Intermediate* addresses a library in layer 2, *EndPoint* a library in Layer 3, respectively. Also, the classes contains cases for above tests, to test the affection of the core system.

6.3.2. Results

Table 13 shows the tests passed by the analysis library. In order to pass a test, all assertions that are made in the corresponding unit tests have to be confirmed. Moreover, those assertions have to be fine grained, defining not only that a relationship exists, but how the relationship is build, i.e. the class instance that maps the relationship. Also, in some tests, structure assertions are made, meaning, that not only a relationship is found, but that the relationship is the only relationship found, or one of three, etc.

Test	Passed
Inheritance	✓
MethodAndMemberVar	✓
Signature	✓
Transitive	✓

Table 13: Tests passed by the analysis library.

The following shows the assertions of the inheritance test, to clarify, what exactly is tested. All other test are structured similar and can be found in Appendix A.

```
new LookupType("testpackage.ITestInterface", InterfaceUnit.class)
```

At first, we have to define the class or interface we want to search for. In the inheritance test, this is the *ITestInterface*. The class *InterfaceUnit* is needed, to determine, that the test is only interested in interfaces with the name *ITestInterface* and not in a class, that maybe shares the same name. After that, an instance of the analysis object can be called, to get a set of nodes, representing the types, that are searched for.

```
Assert.assertTrue(! nodes.isEmpty());
Assert.assertTrue(nodes.size() == 1);
Assert.assertTrue(nodes.get(0) instanceof InterfaceNode);
InterfaceNode node = (InterfaceNode)nodes.get(0);
Assert.assertTrue(node.getType() == NodeType.Interface);
Assert.assertEquals("testpackage.ITestInterface",
    node.getWrappedItemQualifiedName());
```

These assertions express the fact, that only one node is returned by the analysis and that this node is a representation of the searched interface.

```
InterfaceNode iTestInterface = (InterfaceNode) nodes.get(0);
Assert.assertTrue(iTestInterface.getIncomingRelations().size() == 1);
```

This is a structure assertion, and describes the fact, that the interface must have only one relationship.

```
AbstractRelation rel = iTestInterface.getIncomingRelations().get(0);
Assert.assertTrue(rel instanceof ClassRelation);
Assert.assertEquals(1, ((ClassRelation) rel).getTypes().size());
ObjectTypeRelationType type0 = ((ClassRelation) rel).getTypes().get(0);
Assert.assertTrue(type0 == ObjectTypeRelationType.Implements);
```

These assertions define the type of relationship that has to be present, i.e. an *Implementation* without an import.

```
AbstractNode testImpl = rel.getFrom();
Assert.assertTrue(testImpl instanceof ClassNode);
Assert.assertTrue(testImpl.getType() == NodeType.Class);
Assert.assertEquals("TestImpl",
    ((ClassNode)testImpl).getClassUnit().getName());
```

At last, the implementation relationship starts from a node wrapping the class *TestImpl*.

7. Conclusion

> And that's the bottom line, 'cause Stone Cold said so!
>
> – *Stone Cold Steve Austin*

This section summarizes this thesis in Section 7.1 and addresses future work in Section 7.2.

7.1. Summary

In this thesis, the capabilities of CloudMIG Xpress have been improved in two ways. The first is an improvement in Java bytecode support for the generation of a KDM instance. The process is now able to discover system parts, that are only present as a set of class files, like a library, for instance. This was achieved through the integration of a Java decompiler into the transformation chain that creates the KDM instance. To find a suited one for this task, an evaluation of current available decompilers has been performed.

For the evaluation, five criteria have been defined, namely the compatibility with the CloudMIG Xpress platform, license, support, feature coverage and generated code quality. For the last criterion, three categories have been specified, in which the output of a decompiler falls; either the decompiler generates no output, a syntactical correct or a syntactical and semantically correct output. But more often than not, the tested decompilers output falls in between these categories, so that two more categories had to be specified, to describe such cases.

The result of the evaluation determined the Dava decompiler best suited for the task, closely followed by JODE. Further testing was required to determine the overall usefulness of Dava, thus two real existing systems — MyBatis JPetStore and JForum — have been decompiled. Then, random samples from the output have been compared to the original code, with the result, that the output is in most cases equivalent, except for the naming of variables.

The second part of this thesis extends the capabilities of the constraint violation detection mechanism implemented in CloudMIG Xpress. A library was introduced to determine the using context of each relevant entity in a KDM instance; a context, which describes, what entities are used in which way.

The library creates the transitive closure of a KDM instance for relevant entities and relationships. This closure describes the reachability aspect of system parts; for instance, it is able to show, how a part of a user interface, dedicated to display file contents, is connected to the service, that loads the file content form the disc.

To demonstrate the capability of the library, test have been formulated, which cover all relevant use cases of classes and interfaces in the Java programming language. The reason behind those tests is, that they also cover all cases, where a constraint violation due to the use of a prohibited class or interface can occur.

All in all, the improvements presented here enables CloudMIG Xpress to perform a more in-depth analysis of legacy systems. Vital parts of a system, that are only available as bytecode, can now be included in the generation of a KDM instance and further processed. Also, due to the library, a more fine grained detection of constraint violations is possible. Moreover, the output of the analysis can be used in the remodeling process, showing a more detailed view on the interconnections in a system.

7.2. Future Work

The statistics presented in Section 6.2 show, that the decompilation process is not optimal, the needed time in particular. Also, some JARs could not be decompiled, due to internal errors in Dava. Sometimes, a JAR could be decompiled partially, if the error-creating class files have been manually removed. The decompiler thus harbors much improvement. It has to be determined, if the errors can be suppressed or even eliminated through proper configuration; if not, Dava has to be either fixed or replaced with a less error prone decompiler such as JODE.

The dependency analysis library is currently only able to create the transitive closure based on the Java constructs of classes, interfaces and methods. For other programming languages, that have other constructs, the library has to be extended. This has to be done by sub-classing the *AbstractEntityHandler* presented in Section 5.3.3, to create a handler that is capable of handling KDM entities representing such language constructs.

8. Related Work

Good artists copy, great artists steal.

– *Attributed to Pablo Picasso*

8.1. Decompiler Analyisis

Miecznikowski and Hendren give an overview of problems they encountered, while developing Dava, a decompiler for Java with the intend to decompile arbitrary bytecode [32]. Among others, the assignment of types to variables and literals is emphasized. An example given in this paper, is the translation of boolean types into integers. While this is a valid substitute, it hinders the decompilation back to a boolean type.

Van Emmerik performed a survey of Java decompilers in 2003 [45]. He collected a set of nine tests to evaluate current freely accessible decompilers and identified JODE as the best. Based on that survey, Hamilton and Danicic also evaluated Java decompilers [21]. They included new and upgraded versions of decompilers and refined the tests. They results point to Dava, Java Decompiler (in this Thesis named Java Decompiler Project) and JODE as the best decompiler for Java, but no one was able to decompile all of the provided tests.

8.2. Dependency Analysis for Program Understanding

Moonen describes a generic architecture for data flow analysis [33], aiding the reverse engineering process. The main goal of this architecture is to prevent reimplementation of the same data flow analysis algorithms for different programming languages. The proposed solution transforms the language-dependent program into an intermediate representation defined through a data flow representation language. Thus, the algorithms have to implemented only for one language.

Leitch uses dependency analysis to estimate refactoring costs and benefits [29]. The premise of this paper is, that the costs of maintaining a “good” designed software is less than the maintaining costs of “bad” designed software. The costs and benefits for each refactoring activity are determined through the calculation of the refactoring ROI (Return on investment). In this calculation, the dependency analysis is applied in two of three steps. In the first step, a procedure-level dependency analysis is used to construct control and data dependency graphs. Refactoring opportunities are identified with the help of the *Code smells* criteria defined by Fowler [17]. The opportunities then constitute

a refactoring plan, which is used to create two new control and data dependency graphs, showing the state of the system after refactoring. The old and new graphs building the base for the calculation of the ROI by subtracting the estimated maintenance costs of the new system from the estimated costs of the old system.

Eisenbarth et al. presents a semi-automated approach to map features to source code using dependency analysis [14]. A feature is defined as functional requirement implemented in the system. In order to detect the relevant units for a feature, scenarios are defined, that contain the execution of the feature. Dynamic analysis is applied to track the control flow while the system is executed under the scenarios. All units present in such a control flow a collected and the intersection is formed. The result is a set of computational units associated with a feature.

References

[1] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. *ACM SIGPLAN Notices*, 33(5):280–290, May 1998. ISSN 0362-1340. doi: 10.1145/277652.277740.

[2] P. Anderson and T. Teitelbaum. Software inspection using codesurfer. In *In Workshop on Inspection in Software Engineering*, 2001.

[3] N. Anquetil and T. C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance*, 11(3):201–221, May 1999. ISSN 1040-550X. doi: 10.1002/(SICI)1096-908X(199905/06)11:3<201::AID-SMR192>3.0.CO;2-1.

[4] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design Pattern Recovery in Object-Oriented Software. In *Proceedings of the 6th International Workshop on Program Comprehension*, IWPC '98, page 153. IEEE Computer Society, 1998. ISBN 0818685603.

[5] Atanas Neshkov Ltd. DJ Java Decompiler. URL `http://www.neshkov.com/`. Last visited: 01.07.2012.

[6] S. Belur and K. Bettadapura. Jdec: Java Decompiler, 2006. URL `http://jdec.sourceforge.net/`. Last visited: 01.07.2012.

[7] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot. MoDisco: A Generic And Extensible Framework For Model Driven Reverse Engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 173–174. ACM, 2010. ISBN 9781450301169. doi: 10.1145/1858996.1859032.

[8] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010. ISBN 0136108040, 9780136108047.

[9] E. J. Chikofsky and J. H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, Jan. 1990. ISSN 0740-7459. doi: 10.1109/52.43044.

[10] C. Cifuentes. *Reverse compilation techniques*. PhD thesis, Queensland University of Technology, July 1994.

[11] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software – Practice and Experience*, 25(7):811–829, July 1995. ISSN 0038-0644. doi: 10.1002/spe.4380250706.

[12] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, July 2009. ISSN 0098-5589. doi: 10.1109/TSE.2009.19.

[13] E. Dupuy. Java Decompiler Project, 2008. URL `http://java.decompiler.free.fr/`. Last visited: 01.07.2012.

[14] T. Eisenbarth, R. Koschke, and D. Simon. Locating Features in Source Code. *IEEE Transactions on Software Engineering archive*, 29(3):210–224, Mar. 2003. ISSN 0098-5589. doi: 10.1109/TSE.2003.1183929.

[15] T. Ekman and G. Hedin. The jastadd extensible java compiler. *ACM SIGPLAN Notices - Proceedings of the 2007 OOPSLA*, 42(10):1–18, Oct. 2007. ISSN 0362-1340. doi: 10.1145/1297105.1297029.

[16] E. J. C. Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley Publishing, Inc., 1st edition, 2005. ISBN 0764574817, 9780764574818.

[17] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. ISBN 0201485672.

[18] S. Frey and W. Hasselbring. The CloudMIG Approach: Model-Based Migration of Software Systems to Cloud-Optimized Applications. *International Journal on Advances in Software*, 4(3 and 4):342–353, 2011. ISSN 1942-2628.

[19] S. Frey, W. Hasselbring, and B. Schnoor. Automatic conformance checking for migrating software systems to cloud infrastructures and platforms. *Journal of Software Maintenance and Evolution: Research and Practice*, 2012. ISSN 1532-0618. doi: 10.1002/smr.582.

[20] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java TM just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM'04, page 12. USENIX Association, 2004.

[21] J. Hamilton and S. Danicic. An Evaluation of Current Java Bytecode Decompilers. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 129–136. IEEE Computer Society, 2009. ISBN 9780769537931. doi: 10.1109/SCAM.2009.24.

[22] R. Hickey. The Clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, page 1. ACM, 2008. ISBN 9781605582702. doi: 10.1145/1408681.1408682.

[23] J. Hoenicke. Java Optimize and Decompile Environment, 1998. URL `http://jode.sourceforge.net/`. Last visited: 01.07.2012.

[24] R. Kazman, S. G. Woods, and S. J. Carrière. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, WCRE '98, page 154. IEEE Computer Society, 1998. ISBN 0818689676.

[25] G. King. Ceylon project. URL `http://www.ceylon-lang.org/`. Last visited: 20.07.2012.

[26] P. Kouznetsov. JAD Java Decompiler, 1997. URL `http://www.varaneckas.com/jad/`. Last visited: 01.07.2012.

[27] K. Kumar. JReversePro Java Decompiler, 2008. URL `http://jreversepro.blogspot.com/`. Last visited: 01.07.2012.

[28] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Oct. 2011.

[29] R. Leitch and E. Stroulia. Assessing the Maintainability Benefits of Design Restructuring Using Dependency Analysis. In *Proceedings of the 9th International Symposium on Software Metrics*, METRICS '03, page 309. IEEE Computer Society, 2003. ISBN 0-7695-1987-3.

[30] Merriam-Webster, Dictionary and Thesaurus. Thesaurus: Dependency. URL `http://www.merriam-webster.com/thesaurus/dependency`.

[31] J. Miecznikowski. New algorithms for a Java decompiler and their implementation in Soot. Master's thesis, McGill University, Montreal, Feb. 2003.

[32] J. Miecznikowski and L. J. Hendren. Decompiling Java Bytecode: Problems, Traps and Pitfalls. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 111–127. Springer-Verlag, 2002. ISBN 3540433694.

[33] L. Moonen. A generic architecture for data flow analysis to support reverse engineering. In *Proceedings of the 2nd international conference on Theory and Practice of Algebraic Specifications*, Algebraic'97, pages 10–10. British Computer Society, 1997. ISBN 3-540-76228-0.

[34] Mozilla Foundation. Rhino: JavaScript for Java. URL `http://www.mozilla.org/rhino`. Last visited: 20.07.2012.

[35] National Institute of Standards and Technology. The NIST Definition of Cloud Computing, Sept. 2011.

[36] Object Management Group, Inc. Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM). URL `http://www.omg.org/spec/KDM/`. Last visited: 12.01.2012.

[37] M. Odersky. The Scala Language Specification. URL `http://www.scala-lang.org/sites/default/files/linuxsoft_archives/docu/files/ScalaReference.pdf`. Last visited: 20.07.2012.

[38] T. Parsons. *Automatic Detection of Performance Design and Deployment Antipatterns in Component Based Enterprise Systems.* PhD thesis, University College Dublin, Nov. 2007.

[39] R. Pérez-Castillo, I. G.-R. de Guzmán, and M. Piattini. Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems. *Computer Standards and Interfaces*, 33:519–532, Nov. 2011.

[40] prabhu. Java Decompiler, 2002. URL `http://sourceforge.net/projects/dcompiler/`. Last visited: 01.07.2012.

[41] M. G. Rekoff. On Reverse Engineering. *IEEE Transactions On Systems, Man, and Cybernetics*, 15(2):244–252, 1985.

[42] D. J. Scales, K. H. Randall, S. Ghemawat, and J. Dean. The Swift Java Compiler: Design and Implementation. Technical report, HP Labs Technical Reports, 2000. URL `http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-2000-2.html`.

[43] N. Shi and R. A. Olsson. Reverse Engineering of Design Patterns from Java Source Code. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 123–134. IEEE Computer Society, 2006. ISBN 0769525792. doi: 10.1109/ASE.2006.57.

[44] A. Umar. *Application Reengineering: Building Web-Based Applications and Dealing with Legacies.* Prentice Hall, 1997. ISBN 0137500351, 9780137500352.

[45] M. Van Emmerik. Java decompiler tests, 2003. URL `http://www.program-transformation.org/Transform/JavaDecompilerTests`. Last visited: 20.07.2012.

[46] H. Van Vliet. Mocha, the Java Decompiler, 1996. URL `http://www.brouhaha.com/~eric/software/mocha/`. Last visited: 01.07.2012.

[47] R. Veldema, R. A. F. Bhoedjang, and H. E. Bal. Jackal, A Compiler Based Implementation of Java for Clusters Of Workstations. In *IN PROC. OF PPOPP*, 2001.

[48] J. F. H. Winkler and G. Dießl. Object CHILL – an object oriented language for systems implementation. In *Proceedings of the 1992 ACM annual conference on Communications*, CSC '92, pages 139–147. ACM, 1992. ISBN 0897914724. doi: 10.1145/131214.131232.

A. Test Implementations

The following sections contain the assertions for each test defined in Section 6.3.1. Some used methods are omitted, but it should be clear form the context and their name, what they do.

FindITestInterface

A simple test for the discovering of an *implementation* relationship.

Searched type: testpackage.ITestInterface

```
// ITestInterface is found and is the only node in nodes list.
Assert.assertTrue(! nodes.isEmpty());
Assert.assertTrue(nodes.size() == 1);
Assert.assertTrue(nodes.get(0) instanceof InterfaceNode);
Assert.assertTrue(
    ((InterfaceNode)nodes.get(0)).getType() == NodeType.Interface);
Assert.assertEquals("testpackage.ITestInterface",
    ((InterfaceNode)nodes.get(0)).getWrappedItemQualifiedName());

// ITestInterface has only one relation
InterfaceNode iTestInterface = (InterfaceNode) nodes.get(0);
Assert.assertTrue(iTestInterface.getIncomingRelations().size() == 1);

// the relation is an implements relation
AbstractRelation rel = iTestInterface.getIncomingRelations().get(0);
Assert.assertTrue(rel instanceof ClassRelation);
Assert.assertEquals(1, ((ClassRelation) rel).getTypes().size());

ObjectTypeRelationType type0 = ((ClassRelation) rel).getTypes().get(0);
Assert.assertTrue(type0 == ObjectTypeRelationType.Implements);

// the relation is from a class named TestImpl
AbstractNode testImpl = rel.getFrom();
Assert.assertTrue(testImpl instanceof ClassNode);
Assert.assertTrue(testImpl.getType() == NodeType.Class);
Assert.assertTrue(
    ((ClassNode)testImpl).getClassUnit().getName().equals("TestImpl"));
```

FindMemberVar

This test discovers the use of *java.lang.System* as a member of the class *TestClass*.

Searched type: java.lang.System

```
// 'System' is found and only member in list
Assert.assertTrue(! nodes.isEmpty());
```

```
Assert.assertTrue(nodes.size() == 1);
Assert.assertTrue(nodes.get(0) instanceof ClassNode);
Assert.assertTrue(((ClassNode)nodes.get(0)).getType() == NodeType.Class);

// 'System' is only referenced once
ClassNode system = (ClassNode) nodes.get(0);
Assert.assertTrue(system.getIncomingRelations().size() == 1);

// The relation is a using relation
AbstractRelation rel = system.getIncomingRelations().get(0);
Assert.assertTrue(rel instanceof ObjectTypeRelation);
Assert.assertEquals(1, ((ObjectTypeRelation) rel).getTypes().size());
Assert.assertEquals(ObjectTypeRelationType.Uses,
    ((ObjectTypeRelation) rel).getTypes().get(0));

// Caller is 'TestClass'
AbstractNode testClass = rel.getFrom();
Assert.assertTrue(testClass instanceof ClassNode);
Assert.assertTrue(testClass.getType() == NodeType.Class);
Assert.assertTrue(
    ((ClassNode)testClass).getClassUnit().getName().equals("TestClass"));
```

FindLocalVarInMethodUnit

The use of *java.io.File* as a local variable is discovered in this test.

Searched type: java.io.File

```
// 'File' is found and only member in list
Assert.assertTrue(! nodes.isEmpty());
Assert.assertTrue(nodes.size() == 1);
Assert.assertTrue(nodes.get(0) instanceof ClassNode);

ClassNode file = (ClassNode)nodes.get(0);
Assert.assertTrue(file.getType() == NodeType.Class);
Assert.assertTrue(file.getClassUnit().getName().equals("File"));

// 'File' is referenced by 'TestClass' in method 'localVar',
// as import statement, and as part of the call graph (represented
// through a method relation)
Assert.assertTrue(file.getIncomingRelations().size() == 3);

// the interesting part for this test is, that the
// 'File' has a UsingRelation
Assert.assertTrue(
    hasRelation(file.getIncomingRelations(), UsingRelation.class));

UsingRelation rel =
    (UsingRelation) findRelation(
```

```
        file.getIncomingRelations(), UsingRelation.class);

Assert.assertTrue(rel.getParentMethodNode() != null);
Assert.assertTrue(rel.getFrom() instanceof ClassNode);
Assert.assertEquals(
    "TestClass", ((ClassNode) rel.getFrom()).getClassUnit().getName());
```

FindInSignature

A test to discover the use of classes and interfaces in a method signature.

Searched types: java.io.OutputStream, java.lang.StringBuffer

```
// only two nodes are found
Assert.assertTrue(! nodes.isEmpty());
Assert.assertTrue(nodes.size() == 2);

// and have the wanted type
Assert.assertTrue(nodes.get(0) instanceof ClassNode);
Assert.assertTrue(nodes.get(1) instanceof ClassNode);

// both are only referenced one in a using relation
Assert.assertTrue(nodes.get(0).getIncomingRelations().size() == 1);
Assert.assertTrue(nodes.get(1).getIncomingRelations().size() == 1);

Assert.assertTrue(
    nodes.get(0).getIncomingRelations().get(0) instanceof UsingRelation);
Assert.assertTrue(
    nodes.get(1).getIncomingRelations().get(0) instanceof UsingRelation);

// the first is used as a parameter and start point is 'TestClass'
UsingRelation usingRelation =
    (UsingRelation) nodes.get(0).getIncomingRelations().get(0);

Assert.assertTrue(usingRelation.getUnitType() == VariableType.ParameterUnit);
Assert.assertTrue(
    ((ClassUnit)usingRelation
        .getFrom()
            .getWrappedItem()).getName().equals("TestClass"));

usingRelation = (UsingRelation) nodes.get(1).getIncomingRelations().get(0);

Assert.assertTrue(usingRelation.getUnitType() == VariableType.ParameterUnit);
Assert.assertTrue(
    ((ClassUnit)usingRelation
        .getFrom()
            .getWrappedItem()).getName().equals("TestClass"));
```

FindFileInTrickyClass

The use of *java.io.File* as defined in Section 6.3.1 is discovered with this test.

Searched type: java.io.File

```
// 'File' is found and only member in list
Assert.assertTrue(! nodes.isEmpty());
Assert.assertTrue(nodes.size() == 1);
Assert.assertTrue(nodes.get(0) instanceof ClassNode);

ClassNode node = (ClassNode)nodes.get(0);

Assert.assertTrue(node.getType() == NodeType.Class);
Assert.assertEquals("java.io.File", node.getWrappedItemQualifiedName());

// File has only one relationship
Assert.assertEquals(1, node.getIncomingRelations().size());

// relation is a 'Calls' relationship
AbstractRelation rel = node.getIncomingRelations().get(0);
Assert.assertTrue(rel instanceof CallsRelation);

// caller is 'TrickyClass'
MethodNode from = (MethodNode) rel.getFrom();
Assert.assertEquals(
    "tricky.TrickyClass",
    from.getParent().getWrappedItemQualifiedName());
```

Transitive

The following tests simulate the use of libraries. For each simulated library, above tests are repeated. However, the assertions are defined under the transitivity aspect, i.e., they assert, that the using context for classes and interfaces are detectable from the class *StartPoint*.

FindEndPointFromStartPoint

Searched type: transitive.StartPoint

```
// 'StartPoint' is found and only member in list
ClassNode node = assertThatNodeExists(nodes);

// node contains a transitive relation with Endpoint as 'to'
for(AbstractRelation rel : node.getOutgoingRelations())
{
    if(rel instanceof TransitiveRelation)
```

```
    {
        AbstractNode to = rel.getTo();

        if(to.getWrappedItemQualifiedName().equals("endpoint.Endpoint"))
        {
            Assert.assertTrue(true);
            return;
        }
    }
}
```

FindFileFormStartPoint

Searched type: transitive.StartPoint

```
// 'StartPoint' is found and only member in list
ClassNode node = assertThatNodeExists(nodes);
// node contains a transitive relation with File as 'to'
for(AbstractRelation rel : node.getOutgoingRelations()) {
    if(rel instanceof TransitiveRelation) {
        AbstractNode to = rel.getTo();
        if(to.getWrappedItemQualifiedName().equals("java.io.File")) {
            List<AbstractRelation> path =
                ((TransitiveRelation)rel).getPath();
            /* possible path:
             * - UsingRelation StartPoint x Intermediate
             * - ClassRelation Intermediate x Endpoint (import)
             * - UsingRelation Endpoint x java.io.File
             */
            AbstractRelation path0 = path.get(0);
            if(path0 instanceof UsingRelation) {
                if(path0.getTo()
                        .getWrappedItemQualifiedName()
                            .equals("transitive.Intermediate")) {
                              AbstractRelation path1 = path.get(1);

                              if(path1 instanceof ClassRelation) {
                                  if(path1.getTo()
                                      .getWrappedItemQualifiedName()
                                        .equals("endpoint.Endpoint")) {
                                      AbstractRelation path2 = path.get(2);

                                      if(path2 instanceof UsingRelation) {
                                          if(path2.getTo()
                                                .getWrappedItemQualifiedName()
                                                    .equals("java.io.File")) {
                                              Assert.assertTrue(true);
                                              return;
                                          }
    ...
}
```

FindFileFormStartPoint

Searched type: transitive.StartPoint

```
// 'StartPoint' is found and only member in list
ClassNode node = assertThatNodeExists(nodes);
// node contains a transitive relation with Calendar as 'to'
for(AbstractRelation rel : node.getOutgoingRelations()) {
    if(rel instanceof TransitiveRelation) {
        AbstractNode to = rel.getTo();
        if(to.getWrappedItemQualifiedName()
                .equals("java.util.Calendar")) {
            List<AbstractRelation> path =
                ((TransitiveRelation)rel).getPath();
            /* possible path:
             * - UsingRelation StartPoint x Intermediate
             * - UsingRelation Intermediate x Endpoint
             * - UsingRelation Endpoint x java.util.List
             */
            AbstractRelation path0 = path.get(0);
            if(path0 instanceof UsingRelation) {
                if(path0.getTo()
                        .getWrappedItemQualifiedName()
                            .equals("transitive.Intermediate")) {
                    AbstractRelation path1 = path.get(1);
                    if(path1 instanceof UsingRelation) {
                        if(path1.getTo()
                                .getWrappedItemQualifiedName()
                                    .equals("endpoint.Endpoint")) {
                            AbstractRelation path2 = path.get(2);

                            if(path2 instanceof UsingRelation) {
                                if(path2.getTo()
                                        .getWrappedItemQualifiedName()
                                            .equals("java.util.Calendar")) {
                                    Assert.assertTrue(true);
                                    return;
                                }
    ...
}
```

Transitive Inheritance

The detection tests for classes and interfaces, that are extended and implemented, respectively, are the same for layer two and three. The only differences are the names of the classes and interfaces, that are searched for. Thus, two methods are introduced, containing the general structure of the tests and are defined as follows:

Searched type: transitive.StartPoint

```java
private void inheritanceExtendsTest(
    String searchedEntityName, String extender)
    {
        LookupType[] forbiddenTypes =
                new LookupType[] {
                    new LookupType("transitive.StartPoint", ClassUnit.class)
        };

        List<AbstractNode> nodes =
                getAnalysis().parseKdm(getTestModel(), forbiddenTypes, false);

        // 'StartPoint' is found and only member in list
        ClassNode node = assertThatNodeExists(nodes);

        boolean correct = false;

        for(AbstractRelation rel : node.getOutgoingRelations()) {
            if(correct) {
                break;
            }

            if(rel instanceof TransitiveRelation) {
                AbstractNode to = rel.getTo();

                if(to.getWrappedItemQualifiedName()
                        .equals(searchedEntityName)) {
                    AbstractRelation last =
                        ((TransitiveRelation) rel).getPath().getLast();

                    if(last instanceof ClassRelation) {
                        Assert.assertEquals(
                            extender,
                            last.getFrom().getWrappedItemQualifiedName());

                        Assert.assertEquals(
                            ObjectTypeRelationType.Extends,
                            ((ClassRelation) last).getTypes().get(0));
                        correct = true;
                    }
                }
            }
        }

        if(! correct) {
                Assert.fail();
        }
    }
```

Searched type: transitive.StartPoint

```
private void inheritanceImplementsTest(
    String searchedEntityName, String implementer)
    {
        LookupType[] forbiddenTypes =
                new LookupType[] {
                    new LookupType("transitive.StartPoint", ClassUnit.class)
        };

        List<AbstractNode> nodes =
                getAnalysis()
                    .parseKdm(getTestModel(), forbiddenTypes, false);

        // 'StartPoint' is found and only member in list
        ClassNode node = assertThatNodeExists(nodes);

        boolean correct = false;

        for(AbstractRelation rel : node.getOutgoingRelations()) {
            if(correct) {
                break;
            }
            AbstractNode to = rel.getTo();

            if(to.getWrappedItemQualifiedName()
                    .equals(searchedEntityName)) {
                AbstractRelation last =
                    ((TransitiveRelation) rel).getPath().getLast();

                if(last instanceof ClassRelation) {
                    Assert.assertEquals(
                        implementer,
                        last.getFrom().getWrappedItemQualifiedName());

                    IsIn<ObjectTypeRelationType> in =
                        new IsIn<ObjectTypeRelationType>(
                            ((ClassRelation) last).getTypes());

                    Assert.assertThat(
                        ObjectTypeRelationType.Implements,
                        in);
                    correct = true;
                }
            }
        }
        if(! correct) {
                Assert.fail();
        }
    }
```

The tests for layer two and three are now defined as follows:

```
public void inheritanceExtendsLayer2() {
    inheritanceExtendsTest(
    "java.lang.Thread", "transitive.Intermediate");
}

public void inheritanceImplementsLayer2() {
    inheritanceImplementsTest(
        "java.io.Serializable", "transitive.Intermediate");
}

public void inheritanceExtendsLayer3() {
    inheritanceExtendsTest(
        "java.lang.Object", "endpoint.Endpoint");
}

public void inheritanceImplementsLayer3() {
    inheritanceImplementsTest(
        "java.lang.Runnable", "endpoint.Endpoint");
}
```

Transitive Signature

The detection tests for signatures in a transitive way are also the same for layer two and three, that is why a helper method is introduced, that defines the test structure:

Searched type: transitive.StartPoint

```
private void findInSignature(
    String returnEntityName,
    String parameterEntityName,
    String parentClassName) {
        LookupType[] forbiddenTypes =
                new LookupType[] {
                    new LookupType("transitive.StartPoint", ClassUnit.class)
        };
        List<AbstractNode> nodes =
                getAnalysis().parseKdm(getTestModel(), forbiddenTypes, false);
        // 'StartPoint' is found and only member in list
        ClassNode node = assertThatNodeExists(nodes);
        boolean returnCorrect = false;
        boolean parameterCorrect = false;

        for(AbstractRelation rel : node.getOutgoingRelations()) {
            if(rel instanceof TransitiveRelation) {
                AbstractNode to = rel.getTo();

                if(to.getWrappedItemQualifiedName()
                        .equals(returnEntityName)) {
```

```
                    AbstractRelation last =
                        ((TransitiveRelation) rel).getPath().getLast();

                    if(last instanceof UsingRelation) {
                        Assert.assertEquals(
                            parentClassName,
                            last.getFrom().getWrappedItemQualifiedName());

                        Assert.assertEquals(
                            VariableType.ParameterUnit,
                            ((UsingRelation) last).getUnitType());
                        returnCorrect = true;
                    }
                }

                if(to.getWrappedItemQualifiedName()
                        .equals(parameterEntityName)) {
                    AbstractRelation last =
                        ((TransitiveRelation) rel).getPath().getLast();

                    if(last instanceof UsingRelation) {
                        Assert.assertEquals(
                            parentClassName,
                            last.getFrom()
                                .getWrappedItemQualifiedName());

                        Assert.assertEquals(
                            VariableType.ParameterUnit,
                            ((UsingRelation) last).getUnitType());
                        parameterCorrect = true;
                    }
                }
            }
        }

        if((! returnCorrect) || (! parameterCorrect)) {
            Assert.fail();
        }
}
```

The tests are then defined as follows:

```
public void SignatureTestLayer3() {
    findInSignature(
        "java.lang.Appendable",
        "java.math.BigDecimal",
        "endpoint.Endpoint");
}
```

```
public void SignatureTestLayer2() {
    findInSignature(
        "java.awt.Button",
        "java.awt.font.FontRenderContext",
        "transitive.Intermediate");
}
```

B. Attached CD

The attached CD contains the following:

Decompiler

This directory contains the following:

- The sources of for the tests defined in Section 4.3.
- The compiled classes for the tests, separated by the compiler version.
- Decompilers, that have been tested.
- The output of the decompilers, separated by compiler version.
- CloudMIG projects, that have been manipulated, to integrate the decompiler.
- The sources of the libraries of JPetStore and JForum.

KDM-based analysis library

The directory contains the Maven project of the dependency analysis library defined in Section 5, as well as the Java projects, that are the base for the tests cases.

Thesis

Contained within this directory is the digital version of this thesis as well as the source files.