”What Programmers Do with Inheritance in Java”
Replicated on Source Code

Çiğdem Aytekin
cigdem.aytekin2@student.uva.nl

September 26, 2014, 109 pages

Supervisor: Tijs van der Storm
Host organisation: CWI - Centrum Wiskunde & Informatica, http://www.cwi.nl

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Abstract

Inheritance is an important mechanism in object oriented languages. Quite some research effort is invested in inheritance until now. Most of the research work about inheritance (if not all) is done about the inheritance relationship between the types (namely, classes and interfaces). There is also some debate about if inheritance is good or bad, or how much inheritance is useful. Tempero et al. raised another important question about inheritance. Given the inheritance relationships defined, they wanted to know how much of these relationships were actually used in the system. To answer this question, they developed a model for inheritance usage in Java and analysed the byte code of 93 open source Java projects from Qualitas Corpus (which is a curated collection of Java open source projects). The conclusion of the study was that inheritance was actually used a lot in these projects - in about two thirds of the cases for subtyping and about 22 percent of the cases for (what they call) external reuse. They report about 2 % of internal reuse. Moreover, they found out that down-call (late-bound self-reference) was also used quite frequently, about a third of inheritance relationships included down-call usage. They also report that there are other usages of inheritance, but these are not significant.

In this study, we replicate the study of Tempero et al. with one major difference: we analyse the source code of the projects and not the byte code. We use the inheritance model of the original study as-is. We also analyse the same projects, however we obtain them from a different source, namely the compiled version of Qualitas Corpus: Qualitas.class Corpus. Our analysis program is written in the Rascal meta-programming language.

In some cases we obtained similar results as the original study. We conclude that at least 60 % of the cases involve subtyping. We have similar results in total reuse. In some other cases, we have different results. We report a lower median of external reuse than the original study (only 3 %). On the contrary, our internal reuse median is higher than theirs (20 %). Our down-call percentage median is also lower (27 % vs. 34 %). For the other possible uses of inheritance, most of our results are similar to the ones of the original study.

We discuss the possible reasons for the differences elaborately and come up with four major causes: First of all, the content of the analysed projects is different in our case, which may affect the results in all usages of inheritance. For one third of the projects in the Corpus, we analysed different versions. Moreover, the source code distribution contains often different set of types than byte code packages. This does not mean reporting fewer or more cases, but simply a different percentage. We believe that this has the largest impact on results among all the differences in our study set-up. Secondly, the byte code analysis can be misleading when detecting some particular cases of down-call and external reuse, and we suspect that the original study reports more cases for down-call and external-reuse than it should. Thirdly, our technical limitation about analysis of non-system methods may cause a fewer number of reported cases in subtype and external reuse attributes. As the last cause, we suspect the difference in interpretations of some inheritance definitions. Although we communicated with the authors for clarification and got satisfactory answers in many cases, we can not be totally certain that we have the right understanding of each definition without doing an extensive case study together with the authors.

Preface

My name appears as the author of this thesis, but there are many people without the support of whom this work could not be produced.

First of all, I would like to thank my thesis supervisor Tijs van der Storm for letting me conduct this study in CWI and for his support for my work. I also thank to Ewan Tempero, the first author of the original study. He answered all my e-mails with questions about the original work and provided us with detailed answers. Thanks to those e-mails I gained a better understanding of the original study, and this, in turn, increased the quality of the replication study. My special thanks go to Vadim Zaytsev for encouraging me to present my work at SATToSE (the Seminar Series on Advanced Techniques & Tools for Software Evolution) 2014 in L'Aquila, Italy. Thanks to him I joined the seminar and that brought me new insights about my work. I also would like to thank Jurgen Vinju, Davy Landman and Ashim Shahi from CWI sincerely. Jurgen has not only answered all my questions with patience but also has been a source of motivation with his enthusiasm. Davy also spent quite some time answering my Rascal questions. Ashim dealt with almost all the Rascal problems I've encountered: first by listening and understanding what was going on, and then by providing timely fixes whenever I needed. Jorryt-Jan Dijkstra read my thesis in detail and provided me with valuable feedback. Also thanks to Bas Brekelmans, who is also replicating the same study, for his valuable input, especially about validation of my results. I would like to express my appreciation here for all the SWAT (Software Analysis and Transformation) team members in CWI. They are very nice colleagues to work with.

I also thank my daughter Liselot for being a continuous source of motivation and inspiration, also during this study. I could make enough time for this work thanks to the loving support of my husband, Wildrik. I want to thank my parents, I really appreciate their confidence and support throughout my life.

Finally, I want to say that I am very grateful to every teacher and lecturer of me. Without your efforts I could not have come this far...

Chapter 1

Introduction

The subject of this thesis is the various uses of inheritance in Java. The inheritance mechanism in object oriented languages has been subject of many research studies in software engineering. Three researchers, E. Tempero, H.Y. Yang and J. Noble have brought a new perspective to the inheritance research. Many studies about inheritance concentrate on how the inheritance is defined between the types involved. For example, well known metrics about inheritance like DIT (depth of inheritance tree) and NOC (number of children) [CK94] reflect the structure of inheritance. These three researchers, however, asked a new question: given an inheritance structure that is already defined in a project, for which purposes the programmers use the inheritance relationships for?

To answer this question, they thought of various uses of inheritance and presented an inheritance usage model. For example, when a child type issues a call to a method of a parent type, the child type reuses a piece of code of the parent, and this usage is modelled as *reuse*. Moreover, they analysed a corpus of Java open source projects to find out if the defined inheritance relationships were actually used, and if so, how much.

We have written an analysis tool in Rascal meta-programming language to replicate their study. We use the inheritance usage model proposed by the authors as is, and furthermore, also analyse the same projects from their corpus (with some differences in versions). We have a major difference with the original study, we are analysing the source code, while the original study analysed the byte code.

This thesis documents our replication study in detail.

1.1 Problem Statement

We wanted to know if we could verify the results of the original study for inheritance usage. To be able to do the replication properly, we had to have a good understanding of original model and usage metrics which the authors defined. Moreover, acquiring the corpus they are using was also necessary.

Writing the Java source code analysis tool was the biggest part of the work. We have programmed in Rascal. Rascal has libraries for many functionalities needed for Java source code analysis (like building an abstract syntax tree) and therefore was the ideal choice for us.

1.2 Motivation

Why did we choose to do a replication study about inheritance usage?

First, we want to explain why we've chosen for replication. Replication studies play an important role in empirical scientific studies. Brooks et al. explain this in their chapter "Replication's Role in Software Engineering" [BRW+08]: "By other researchers successfully repeating an experiment, confidence is built in the procedure and the result. Without the confirming power of external replications, a result should be best regarded as of limited importance and at worst with suspicion and mistrust". Replication in software engineering is also important: "Without the confirming power of external replication, many principles and guidelines in software engineering should be treated with caution".

If replication is this important in software engineering, it is fair to expect many replication studies in the literature. However, the converse is true: "A systematic survey of controlled experiments in software engineering between 1993 and 2002 by Sjoberg et al. (2005) [SHH+05] found only twenty studies claiming to be replications of which only nine were external replications."

We see that the role of replication is very important for empirical software engineering research and we also see that there are very few replication studies. We have chosen replication as an answer to this need.

Why did we choose the inheritance usage? Inheritance is an important mechanism of object oriented languages and has been the subject of many studies until now. According to Taivalsaari [Tai96] inheritance is often qualified as the distinguishing feature of object oriented languages. Many benefits of object oriented programming, like improved conceptual modelling and reusability are accredited to inheritance. Taivalsaari also notes that although inheritance plays a central role in object oriented paradigm, there are many questions about its definition and usage.

Many empirical studies until now concentrated mainly on the definition of the inheritance relationship. The metrics such as DIT (depth of inheritance tree) and NOC (number of children) which are used in the studies about object oriented program analysis are also about how the inheritance tree is defined. The study we replicate brings a different perspective to the inheritance research by investigating the actual usage of already defined inheritance relationships in the code. The authors observe inheritance solely from *usage* perspective and define a model to represent different uses of inheritance in the code.

In this respect the original study addresses an important subject which still needs to be investigated. In addition to that, the authors shed light on a not yet explored area of inheritance; namely its actual usage in the code.

1.3 Organization of This Thesis

After this introduction, we first look into the related work in chapter 2. Thereafter we explain the original study briefly; the research questions, artefacts, limitations and finally the results. Then follows the definitions chapter which explains the original inheritance model definitions with examples. This chapter is followed by the chapter about metrics; having defined the various usages of inheritance, how to measure the usage in detail. Definition and metrics chapters give the necessary information to be able to understand the details of both studies. We continue then with the chapter about the replication study, especially how it differs from the original study. Chapter 7 explains our Rascal implementation for each category of inheritance usage briefly. The challenging parts of our implementation are presented in the subsequent chapter, chapter 8.

Our limitations, most of which were introduced by limited amount of time to complete the thesis work, are listed in the subsequent chapter. We present our results in chapter 10. An extensive discussion about our results is given in chapter 11. In chapter 12, we explain the possible threats we identified for validity. The following chapter discusses our results with respect to results of the original study. We present our ideas about future work in chapter 13. We end the thesis with a conclusion: chapter 14.

When organizing this thesis, the suggestions from the article "Reporting Guidelines for Controlled Experiments in Software Engineering" [Car10] are used as guidelines.

Chapter 2

Related Work

An extensive study about the "notion of inheritance" is done by Taivalsaari in 1996 [Tai96]. In this survey study he explains different purposes for which inheritance is used and gives different classifications of inheritance. This work has been mentioned many times in the original article and two research questions refer to the down-call (late-bound self-reference) and subtype concepts as explained in this article. Taivalsaari successfully clarifies the difference between certain inheritance concepts which are closely related to each other. Especially the distinction between the subclassing ("subclassing is an implementation mechanism for sharing code and representation"), subtyping ("subtyping is a substitutability relationship: an instance of a subtype can stand for an instance of its super type") and Is-a ("Is-a is a conceptual specialization relationship: it describes one kind of object as a special kind of other") is remarkable. He also mentions that two major benefits of inheritance, reusability and conceptual modelling are in fact opposite goals: "in order to obtain maximum reusability, one usually has to sacrifice modelling, and vice versa."

A large scale empirical study about Java inheritance is done by Tempero, Noble and Melton (as mentioned before Tempero and Noble are also the authors of the study we are replicating) and is published in 2008 [TNM08]. They used an earlier version of Qulaitas Corpus, which is also used in our replication study. This work concentrates more on the definition of inheritance relationships between the Java types, rather than the actual usage of inheritance. The authors found out some characteristics of how types are defined with respect to inheritance in the corpus: most classes in Java programs are defined using inheritance from other "user defined" types (the types which are not defined in Java Standard API or libraries used by the project, but in project itself), classes and interfaces are used in different ways, with approximately one interface defined for every ten classes, most types are relatively shallow in the inheritance hierarchy, almost all types have fewer than two types inheriting from them (but there are some very popular types, and many types will inherit from them) and larger systems proportionally make more use of inheritance from user defined classes and less use of standard or third-party library classes.

Nasseri, Counsell and Shepperd made a study about the evolution of inheritance in Java open source systems (OSS) [NCS08]. They observed the evolution of seven Java OSS in time and concluded that inheritance hierarchy grows 'breadth-wise' rather than 'depth-wise'. They saw that 96 % of all classes added in time were added at the inheritance depth of 1 or 2. This study uses four object oriented metrics, two of which we mentioned before (DIT - depth of inheritance tree and NOC - number of children). The other two metrics they use is from Henderson-Sellers [HS95] , Specialization Ratio (SR) and Reuse Ratio (RR). Specialization ratio is equal to number of subclasses divided by number of super classes and reuse ratio is the number of super classes divided by total number of classes. All of these metrics are concerned about the definition of inheritance relationships, and not about the 'usage' as in our original study.

The study of Lämmel, Linke, Pek and Varanovich [LLPV11] analyses a corpus of .NET projects for the usage of .NET API in the source code. Their method involves some usage metrics, in the sense that they also investigate if a type is actually referenced or bound late in the project code. With referencing they mean that the type should actually be (re)used in the project, and with bound late they mean, in their own words, "if there is a method call with the framework type as static receiver type and a project type as runtime receiver type." They analysed byte code of the systems and their corpus consists of 17 projects. Their usage results are remarkable, however we should keep in mind that they conducted the study from API usage perspective. Among all the classes defined, they found that 10 % of all classes were referenced. The interface reference percentage (among all interfaces) is also similar: 11,8 %. They found that only 2,6 percent of all specializable types were actually specialized and 2,1 percent of all specializable types were bound late.

The studies mentioned above analysed corpora of projects. Some other empirical studies concentrated on the effect of inheritance on the quality of code, especially on the maintainability of code. Instead of analysing code, these studies made experiments with programmers, and measured programmers performance (for example, the time it takes them to make a certain change to the code).

An early study of Mancl and Havanas from 1990 [MH90], focuses on the usage of C++ as programming language on software maintenance. They have measured the maintenance efforts (measured in lines of changed code, number of changed files and the amount of different code blocks that changed) needed for modification requests on the software system they were using. The system has used three major programming paradigms; conventional structured design, abstract data types and object oriented design. The authors compared the maintenance effort needed for structured design and object oriented design (which involves the usage of inheritance). They also investigated the number of required interface changes for each modification request. The study reports that the object oriented parts of their system have produced several maintenance benefits. First of all, software reuse has doubled as more of the systems is built in an object oriented way. Secondly, the new features are added to the object oriented sections with less effort and they caused fewer interface changes within the system. Lastly, the adaptations to external changes in their system's object oriented modules cause significantly fewer source lines to be modified than for similar changes in the structured modules.

Another study with programmers has been carried out by Daly, Brooks, Miller, Roper and Wood in 1996 [DBM+96]. They did a series of experiments with C++ code and came up with two results. First of all, they concluded that the programmers completed maintenance tasks quicker in a class hierarchy with depth three when compared with a flat hierarchy (no subclassing at all). Their second conclusion showed that if the depth increased more, it had the contrary effect. Namely, the subjects maintaining the object-oriented software with five levels of inheritance depth took longer than those with a flat hierarchy.

Cartwright [Car98] replicated the study done by Daly et al. and, unexpectedly, found out the opposite of their results in 1998. Their study also involved a very similar maintenance task which is conducted on C++ code. Just like in the original study, they also compared the maintenance times of the code with three levels of inheritance with zero levels of inheritance. In Cartwright's own words: "It was found that subjects took significantly longer to make the same change on the program with inheritance, however, their changes were more compact."

After having discussed the related work, we will describe the study we replicate in the following chapter.

Chapter 3

The Original Study

3.1 Overview

Ewan Tempero, Hong Yul Yang and James Noble have published the article ”What Programmers Do with Inheritance in Java” in The European Conference on Object Oriented Programming (ECOOP) in 2013 [TYN13]. The aim of our work is to replicate the study on which their article is based. In this chapter, we explain the original study. After giving a short introduction, we present the research questions, used artefacts, limitations and the results of the original study. The detailed explanation of the definitions used in the inheritance model is essential for understanding the original study (and this study), and therefore it deserves a chapter of its own (4).

As mentioned in the previous chapter, Tempero et al. conducted research about Java inheritance also before this study. Their previous inheritance research concentrated on the existing inheritance relationships in Java projects [TNM08]. This study is different in the sense that they wanted to find out for which purposes inheritance was actually used in Java. In their own words: ”having made the decision to use inheritance at the design level, what benefits follow from the use of inheritance?”

The difference between defining an inheritance relationship and actually using the inheritance relationship in Java can be explained with the following example. Consider two classes Q and P. If Q is defined as a subclass of P, then we talk about an inheritance relationship between Q and P. Let us assume that a method is defined in P and it is not overridden in Q (say, method p()). If the method p() is actually called on an object of type Q, we talk about an inheritance usage because in this case, a piece of code which is defined in P is actually re-used on an object of type Q.

The authors make three contributions with their study. First of all, they introduce a model in which different usages of Java inheritance are defined. They analyse a corpus of Java projects [TAD$^+$10] using this model. Their second contribution is that they make their study replicable. The analysed corpus [TAD$^+$08] and the analysis results [TYN08] are available. Finally, they present their study results which show that inheritance is used quite considerably in open source Java projects, especially for subtyping and for what they call external reuse.

For convenience we will use the abbreviation WhaPDJI (What Programmers Do with Java Inheritance) when we refer to the original study in the rest of this document.

3.2 Research Questions

The following are the research questions of the original study:

Research Question 1: To what extent is late-bound self-reference relied on in the designs of Java Systems? The terms late-bound self-reference and down-call are synonyms in the study and are defined in 4.8

Research Question 2: To what extent is inheritance used in Java in order to express a subtype relationship that is necessary to design? Subtype usage happens when an object of descendant type is supplied where an object of ascendant type is expected. Detailed definition for subtype can be found in 4.7

Research Question 3: To what extent can inheritance be replaced by composition? The authors wanted to know if there were opportunities for replacing inheritance by composition. The inheritance relationships which involve reuse, but which do not involve subtype use, are candidates for such a replacement. The two different types of reuse, internal and external, are defined in 4.5 and 4.6 respectively.

Research Question 4: What other inheritance idioms are in common use in Java systems? The authors identified some other uses of inheritance in addition to the already mentioned ones. These are explained in 4.9.

3.3 Artefacts

The authors make use of the following artefacts:

Qualitas Corpus The Qualitas Corpus is a curated collection of software systems intended to be used for empirical studies of code artefacts [TAD+10]. In the original study, 93 different open source Java projects are used for the code analysis from the 20101126 release of the Qualitas Corpus. The authors also made a longitudinal study on two projects: project ant (20 releases in total, from release 1.1 to 1.8.1) and project freecol (23 releases in total, from 0.3.0 to 0.9.4). Corpus website is: [TAD+08]

The Qualitas Corpus contains both byte and source codes of the projects.

Study Results Web Page In addition to the results documented in their article, the authors also have placed a package of detailed information on a web site [TYN08]. They document here the definitions and metrics they have used for the study as well as the results of their measurements on the Corpus projects. The definitions and metrics used in the original study can be found in chapters 4 and 5 respectively.

Analysis Tool The programs which are used to analyse the byte code. Not much information is given about the tool in the WhaPDJI study. We know that it is based on Soot framework (a Java optimization framework) [VRCG+99] and that because of the memory limitations of the tool, they could not analyse a few big projects from Qualitas Corpus.

3.4 Limitations of the Original Study

The limitations of the original study are as follows:

- The study is limited to Java classes and interfaces only. Exceptions, enums and annotations are excluded,
- The third party libraries are not analysed. Because of this, they can not determine the purpose of some inheritance relationships. They introduced the framework attribute for the cases which subtype usage is suspected in this context,
- The inheritance relationships between system types and non-system types are not modelled. Please see section 4.1 for definition of system type,
- Heuristics are used when defining framework and generics attributes. For definitions of these two attributes, please see 4.9,
- The authors use the Java byte code as input to their analysis tool, byte code may in some cases incorrectly map to source code. The authors state that this occurs very rarely but when it occurs, it may affect model fidelity. The original article provides detailed explanation about this limitation in section 4.3 (Analysis Challenges). They give an example about how a false negative may occur because of this issue,

- They do make static code analysis and this may have impact on their down-call results, the results may be overstating the reality. This limitation is further explained in the section about down-call 4.8.

3.5 Results

For Research Question 1: They conclude that late-bound self-reference plays a significant role in the systems they studied - around a third (median 34 %) of CC (class class) edges involve down-calls. For definition of CC attribute, please see 4.3,

For Research Question 2: At least two thirds (median 66 %) of all inheritance edges are used as subtypes in the program, the inheritance for subtyping is not rare,

For Research Question 3: The authors found that 22 % or more edges use external re-use (without subtyping) and 2 % or more use internal re-use (without subtyping or external reuse). They conclude that this result signals opportunities for replacing inheritance with composition,

Research Question 4: They report quite a few other uses of Java inheritance (constant, generic, marker, framework, category and super), however the results show that big majority of edges (87 %) in their Corpus can already be explained with one of the subtype, external re-use, internal re-use uses and the other uses of inheritance in Java are rare.

Chapter 4

Definitions

Definitions are very important for this study. They are used extensively in the metrics and to be able to interpret the metrics correctly, understanding the definitions is essential.

The authors of the WhaPDJI study model inheritance relationships in a graph. The descendant ascendant types are modelled as vertices of the graph and inheritance relationships as edges. The authors also talk about the different attributes of the edges (like a CC edge, or a subtype edge, etc....). Although this is a good way of modelling inheritance, we preferred to refer to an edge as an ordered relationship between two types: <descendant,ascendant> . The reason for this choice was to introduce our terminology independent of an implementation choice.

When we use *type* in a definition, it may be a Java class or a Java interface. If a definition is only meaningful for a class or an interface, however, we use specifically *class* or *interface.*

4.1 System Type

A system type is created for the system under investigation. A non-system type or an external type, on the other hand, is used in the system, but it is not defined in the system. In the following Java code example, the class G is a system type and ArrayList is a non-system type.

```
import java.util.ArrayList;

public class G extends ArrayList { }
```

4.2 User Defined Attribute

The descendant ascendant pair in an inheritance relationship has user defined attribute if both the descendant and ascendant are system types. In the example, pair <Q,P> has the user defined attribute, while pair <L, ArrayList> has not.

```
class P{ }

class Q extends P {  }

import java.util.ArrayList;
class L extends ArrayList;
```

4.3 CC, CI and II Attributes

The descendant ascendant pair in an inheritance relationship in Java can have one of the three attributes: CC (Class Class) - both descendant and ascendant are classes, CI (Class Interface) -

descendant is a class and ascendant is an interface or II (Interface Interface) - both descendant and ascendant are interfaces. In the example, the pair <Q,P> has the CC attribute and the pair <Q,I> has the CI attribute.

```
interface I {}

class P{ }

class Q extends P implements I {}
```

4.4 Explicit Attribute

The inheritance relationship is qualified as explicit if it is described directly in the code. In the example, pairs <C, P> and <G, C> have explicit attribute. <G,P> however, does not have explicit attribute. Although there is an inheritance relationship between G and P, it is only implied, and not defined explicitly in the program.

```
class P{ }

class C extends P { }

class G extends C {  }
```

4.5 Internal Reuse

Internal reuse happens when a descendant type calls a method or accesses a field of its ascendant type. In the example, child class Q accesses parent method p() and parent field pField in method q(). The reuse may also occur on an object of child type. The call aQ.p() qualifies also as internal reuse, because this access takes place in the child class Q itself.

```
public class P {
    public int pField = 0;
    void p() {
    }
}

public class Q extends P {
    void q() {
        p();          // internal reuse via method call
        pField = 1;   // internal reuse via field access
        Q aQ = new Q();
        aQ.p();      // internal reuse via method call
    }
}
```

4.6 External Reuse

External reuse is like internal reuse, except for that the access to a method or a field of the ascendant type happens not within the descendant type itself, but in another type. The access still happens on an object of descendant type. According to the original study, the class in which the external reuse occurs may not have any inheritance relationship with the descendant or ascendant type. We have chosen to relax this restriction, because with this definition, the reuse which occurs in a class in the inheritance hierarchy elsewhere than the descendant itself would not be counted at all. In the example, a reuse occurs in the method t() of class T. We classify this external reuse because the reuse does not occur in the child type (Q).

```
public class P {
    public int pField = 0;
    void p() {
    }
}

public class Q extends P { }

public class E {
    void e() {
        Q aQ = new Q();
        aQ.p();          // external reuse via method call
        aQ.pField = 1;   // external reuse via field access
    }
}

public class T extends Q {
    void t() {
        Q aQ = new Q();
        aQ.p();         // external reuse via method call
    }
}
```

4.7 Subtype

Subtype usage happens when an object of descendant type is supplied where an object of ascendant type is expected. The original study states that subtype usage can occur in four occasions: when assigning object(s), during parameter passing, when returning an object in a method or casting an object to another type. Contrary to internal and external reuse, the place where the subtyping occurs is not of any importance here. Note that subtyping can also occur in an enhanced for loop (for each loop) or in a ternary operation in Java. Therefore these two constructs should also be included in the analysis.

```
public class T { }

public class S extends T { }

import java.util.ArrayList;
public class X {
    S anS;

    void a(T aT) {     }

    T b() {
        return anS;                          // return statement
    }

    void x() {
        T aT = new S();                      // assignment
        a(anS);                              // parameter passing
        T anotherT = (T)anS;                 // casting
        ArrayList<S> aList = new ArrayList<S>();
        for (T anE : aList) {                    // enhanced for loop
            // ...
        }
                T aT = (2 < 3) ? new S() : new T();  // ternary operator
    }
}
```

There are three cases of subtyping which need additional explanation. Firstly, the casting can occur in two ways, as up-casting and down-casting. Secondly, an interesting construct - sideways casting - can result in subtype usage. Finally, in some occasions the `this` reference can change type and we can again talk about a subtype usage. These cases are defined separately in the subsections below.

4.7.1 Up-casting and Down-casting

Casting can occur in two directions in Java, as explained in [SB08]. If an instance is cast to a type which is higher in the inheritance hierarchy (one of the ascendant types), then we talk about up-casting. Down-casting happens if an instance is cast to a type which is lower in the inheritance hierarchy (one of the descendant types). These two types of casting are depicted in the following example:

```
public class P { }

public class C extends P { }

public class GC extends C { }

public class N {
    void n() {
        GC aGC = new GC();
        P aP = (P)aGC;       // up-casting
        aGC = (GC)aP;        // down-casting.
    }
}
```

In our analysis, both cases of casting are marked as subtype usage.

4.7.2 Sideways Casting

Sideways casting is an interesting case which results in subtype usage between a class and two interfaces. The following example is taken as is from the original study. The cast in the method demo() will be successful if an instance of SidewaysC is passed to the method as parameter. The pairs <SidewaysC, SidewaysA> and <SidewaysC, SidewaysB> both get the subtype attribute.

```
public interface SidewaysA { }
public interface SidewaysB { }
public class SidewaysC implements SidewaysA, SidewaysB { }
public class Sideways {
    public void demo(SidewaysA sa) {
        SidewaysB sb = (SidewaysB) sa;
    }
}
```

4.7.3 This Changing Type

Another instance of subtype usage in Java occurs when `this` reference causes a type change. In the following example, when class C is instantiated, the initializer of its ascendant class (P) is called.

The constructor of class A expects a parameter of type P, but `this` reference in the `new A(this)` statement will be of type C this time.

```
public class P {
    private A anA = new A (this);
}

public class C extends P { }
```

In addition to the example given, `this` reference can also change type if it is passed as a parameter to a method which expects the parent type. In the following example:

```
public class N {
    void expectAP(P aP) {        }
}

public class P {
    void p() {
        N anN = new N();
        anN.expectAP(this);  // this will change type if p()
    }                        // is called on an object of type C
}

public class C extends P { }

public class X {
    void x() {
        C aC = new C();
        aC.p();
    }
}
```

`this` reference will only change type if method p() is issued on an object of a descendant type of P. In our example, this descendant is class C.

When doing our analysis, we inspect both cases. In the first case, we do search for instantiation of an object of type C in the source code. And in the second case, we give the subtype attribute to relation <C,P> only if the method p() is called on an object of type C. We explicitly search for such a method call in the source code.

4.8 Down-call

The terms down-call and late-bound self-reference have the same meaning in the original study. Down-call refers to the case when a method in the ascendant type (ascendant-method) makes a call to another method (descendant-method) which is overridden by the descendant type. When an object of descendant type calls the ascendant-method, the descendant-method of the descendant type will be executed. This case is called *down*-call, because a descendant type is found under the ascendant type in the inheritance hierarchy.

```
public class P {
    void p() {
        q();
    }
    void q() {
    }
}

public class Q extends P {
    void q() {
    }
}

public class D {
    void d() {
        Q aQ = new Q();
        aQ.p();      // when p() is executed,
    }                //  Q#q() is called instead of P#q()
}
```

From the article and the study results we could not decide if authors looked for the actual call (in our example `aQ.p()`) to happen. We have asked this to the authors per e-mail [Tem14], and learned that they do not necessarily look for the actual call. In our example, the definitions of P and Q will be enough for them to get the down-call attribute and whether `p()` is called on an object of type Q does not matter.

4.9 Other Uses of Inheritance

Next to reuse, subtype and down-call, the authors also defined other uses of inheritance: category, constants, framework, generic, marker and super. As we will see in the results section, authors state that these other uses of inheritance occur much less frequently than reuse and subtype uses. We will explain these uses in detail in this section.

4.9.1 Category

Category inheritance relationship is defined for the descendant ascendant pairs which can not be placed under any other inheritance classification. (We should also note that for this definition, ascendant type should be direct ascendant of the descendant type, i.e. no type should have been defined between the two types in the inheritance hierarchy.) In this case, we search for a sibling of the descendant which has a subtype relationship with the ascendant. If we can find such a sibling, we assume that the ascendant is used as a category class, and the descendant is placed under it for conceptual reasons. In the example shown, S has subtype relationship with P, and C and S are siblings. If no other inheritance usage is found between C and P, then their relationship is classified as category.

```
public class P { }

public class C extends P { }

public class S extends P { }

public class R {
    void r() {
        P aP = new S();      // subtype usage between S and P
    }
}
```

4.9.2 Constant

A descendant ascendant pair has constant attribute if the ascendant only contains constant fields (i.e., fields with `static final` attribute). Furthermore, the ascendant should either have no ascendants itself or if it has ascendants, these ascendants should only contain constant fields themselves. In the example, the pair <B,A> has constants attribute.

```
public class A {
    public static final String c = "";
    static final boolean b = true;
    static final double d = 2.2;
    static final float f = 3.3f;
}

public class B extends A {
    // fields and methods of B...
}
```

4.9.3 Framework

A descendant ascendant pair will have the framework attribute if it does not have one of the external reuse, internal reuse, subtype or down-call attributes and the ascendant is a direct descendant of a third party type. Moreover, the first type should be a direct descendant of the second type. In the example, <H,G> pair has Framework attribute. Note that this attribute is used as a heuristic to classify suspected subtype usage. The authors needed such a heuristic because they do not analyse the third party or standard Java API libraries.

```
import java.util.ArrayList;
public class G extends ArrayList { }

public class H extends G { }
```

4.9.4 Generic

Generic attribute is used for the descendant ascendant (for example : descendant type R, and ascendant type S) pairs which adhere to the following:

1. S is parent of R. (i.e. S is direct ascendant of R.)
2. R has at least one more parent, say, T.
3. There is an explicit cast from java.lang.Object to S.
4. There is a subtype relationship between R and java.lang.Object

Generic definition is used to model a case which occurs in collections that are instantiated without type parameters (raw types). An object of type R can be saved in an object of type T, and placed in the container. When the same object is retrieved from the collection, it can be cast to type S.

We have a limitation about the generic attribute. The first three items listed above should also hold for our study, while we do not analyse the fourth rule (there is a subtype relationship between R and object). When doing the subtype analysis, we do not specifically need to look at the subtype relations between user defined classes and java.lang.Object. Because of time limitations, we did not want to introduce this additional step in our subtype analysis. We do not expect this limitation to have big consequences for the whole study, because the generic inheritance usage is part of the other uses of inheritance, and is not expected to occur frequently. This limitation is also discussed in chapter 9.

4.9.5 Marker

Marker usage for a descendant ascendant pair occurs when an ascendant has nothing declared in it. Moreover, just like the constants definition, the ascendant should either have no ascendants itself, or if it has ascendants, its own ascendants should have nothing declared in them. Ascendant should be defined as an interface and descendant may be a class or an interface.

```
public interface H { }

public class G implements H {
    void g() { }
}
```

The marker interfaces are generally used for conceptual classification of classes in Java.

4.9.6 Super

A descendant-ascendant pair will qualify for super attribute if a constructor of descendant type explicitly invokes a constructor of ascendant type via `super` call.

```
public class L {
    public L() {
    }
}
public class K extends L {
    public K () {
            super();
    }
}
```

4.10 Direct and Indirect Usage

The concepts of direct and indirect usage have important consequences for the analysis results. The WhaPDJI study concentrates on the explicit inheritance relationships (that is, the inheritance relationship is explicitly defined by the programmer in the code). However, external reuse as well as subtype usage may occur between the types which have implicit inheritance relationship, as in the following example:

```
public class P {
    void p() {
        // ....
    }
}

public class C extends P { }

public class GC extends C { }

public class N {
    void n() {
        GC aGC = new GC();
        aGC.p();          // external reuse, between GC and P
        P aP = aGC;       // subtype usage between GC and P
    }
}
```

In WhaPDJI study, such an indirect usage between a descendant (in our example GC) and an ascendant (P) causes all inheritance chain between the descendant and the ascendant to receive that usage attribute. In the example, both the pair <P,C> and <C,GC> will be marked with subtype and external reuse attributes. If the inheritance hierarchy between the descendant and ascendant contains many levels, one external reuse and/or subtype usage will cause all the pairs in the hierarchy to be marked as usage.

If the usage occurs between two types which have explicit relationship, this usage is said to be direct and causes no additional pairs to be counted as subtype or external reuse.

It is also important to note that this concept is defined for external reuse and subtype usage only. For all other types of inheritance, such a distinction is not made, and no extra inheritance pairs are added to the results.

4.11 Inheritance Usage vs. CC, CI and II Relationships

It is useful to see which kind of inheritance usage can occur in which kind of type pairs. For example, subtype usage can be seen in all class-class, class-interface and interface-interface relationships. Down-call is only defined for CC relations, whereas the ascendant type in the marker usage can only be an interface. The table 4.1 lists the possible combinations:

	CC	CI	II
Internal Reuse	✓	✓	✓
External Reuse	✓	✓	✓
Subtype - General	✓	✓	✓
Subtype - Sideways Casting	X	✓	✓
Subtype - This Changing Type	✓	✓	X
Down-call	✓	X	X
Category	✓	✓	✓
Constants	✓	✓	✓
Framework	✓	✓	✓
Generic	✓	✓	✓
Marker	X	✓	✓
Super	✓	X	X

Table 4.1: Possible inheritance usages in class-class, class-interface and interface-interface pairs.

Because of time limitations, we did not implement the analysis of following relationships:

- In class-interface and interface-interface pairs, internal reuse is not analysed,
- This changing type analysis is not made for class-interface pairs,
- Category relationship is not analysed for interface-interface pairs.

These limitations are also reported in chapter 9.

Chapter 5

Metrics

After giving the inheritance model definitions in the previous chapter, we will now talk about the metrics used for the code analysis.

The authors of the WhaPDJI study defined the metrics to be able to answer their research questions: How often is down-call used in the class class pairs? How often do we see subtype and reuse? What is the role of the other inheritance relations in projects?

The metrics used in the WhaPDJI study are explained elaborately in the website of the original study [TYN08]. The categories that are used in the graphs of the original article are based on these metrics. For example, Figure 12 of the article depicts various uses of inheritance on CC edges: subtype(ST), external reuse and no subtype (EX-ST), internal reuse only (INO) . The abbreviations ST, EX-ST and INO in the article do correspond to three CC metrics used in the study respectively (perCCSubtype, perCCExreuseNoSubtype and perCCUsedOnlyInRe). Although it was mostly possible to infer these correspondences from the abbreviations, we also e-mailed with the first author (E. Tempero) [Tem14] to acquire the exact relationship between the metrics and the abbreviations. The metrics we use in our study combines all these sources of information (the article, the study and the e-mails with the author).

5.1 Class Class (CC) Metrics

The metrics about the CC (class-class) inheritance relations are explained in table 5.1. For all metrics it holds that the descendant ascendant pair should have explicit and user defined attributes.

Here, it is important to note that the three metrics perCCSubtype, perCCExreuseNoSubtype and perCCUsedOnlyInRe have a different denominator than the other CC metrics, namely numCCUsed. For example, if a project has 87 % of subtype percentage (perCCSubtype), this means that 87 % of the used inheritance relationships (numCCUsed) has subtype attribute. The percentage of the subtype usage for all defined CC relations (numExplicitCC) is not used as a metric in the study.

`numExplicitCC`	Number of CC pairs.

`numCCUsed`
 	
`perCCUsed`
	Number of CC pairs for which some subtype, internal reuse or external reuse was seen.
	`numCCUsed / numExplicitCC`.

`numCCDC` `perCCDC`	Number of CC pairs for which down-call use was seen. `numCCDC / numExplicitCC`
`numCCSubtype` `perCCSubtype`	Number of CC pairs for which subtype use was seen `numCCSubtype / numCCUsed`

`numCCExreuseNoSubtype`
 	
`perCCExreuseNoSubtype`
	Number of CC pairs which do not have the subtype attribute, but which do have the external reuse attribute.
	`numCCExreuseNoSubtype / numCCUsed`

`numCCUsedOnlyInRe`
 	
`perCCUsedOnlyInRe`
	Number of CC pairs which have neither the subtype nor the external reuse attribute, but which do have the internal reuse attribute.
	`numCCUsedOnlyInRe / numCCUsed`

`numCCUnexplSuper`
 	
`perCCUnexplSuper`
	Number of CC edges which do have super use and do not have any other types of inheritance usage.
	`numCCUnexplSuper / numExplicitCC`

`numCCUnexplCategory`
 	
`perCCUnexplCategory`
	Number of CC edges which do have category use and do not have any other types of inheritance usage.
	`numCCUnexplCategory / numExplicitCC`

`numCCUnknown`
 	
`perCCUnknown`
	Number of CC edges which do have an inheritance relationship, but which do not have any of the inheritance usage attributes defined.
	`numCCUnexplCategory / numExplicitCC`

Table 5.1: Class Class Metrics

5.2 Class Interface (CI) Metrics

Table 5.2 explains the metrics about CI (class interface) inheritance relations. Just like CC metrics, for all CI metrics it holds that the descendant ascendant pair should have explicit and user defined attributes.

`numExplicitCI`	Number of CI pairs.
`numOnlyCISubtype` `perOnlyCISubtype`	Number of CI pairs for which subtype use was seen `numOnlyCISubtype/ numExplicitCI`

`numExplainedCI`
 	
`perExplainedCI`
	Number of CI edges which do not have subtype or category inheritance usage but do have some other attribute (one of reuse, framework, generic, marker or constants).
	`numExplainedCI/ numExplicitCI`

`numCategoryExplCI`
 	
`perCategoryExplCI`
	Number of CI edges which do have category use and do not have any other types of inheritance usage.
	`numCategoryExplCI/ numExplicitCI`

`numUnexplainedCI`
 	
`perUnexplainedCI`
	Number of CI edges which do have an inheritance relationship, but which do not have any of the inheritance usage attributes defined.
	`numUnexplainedCI/ numExplicitCI`

Table 5.2: Class Interface Metrics

5.3 Interface Interface (II) Metrics

II (Interface Interface) metrics are depicted in table 5.3. Just like CC and CI metrics, only the pairs which are explicit and user defined are taken into account.

`numExplicitII`	Number of II pairs.
`numIISubtype` `perIISubtype`	Number of II pairs for which subtype use was seen `numIISubtype/ numExplicitII`
`numOnlyIIReuse` `perOnlyIIReuse`	Number of II pairs for which external use was seen `numIIReuse/ numExplicitII`

`numExplainedII`
 	
`perExplainedII`
	Number of II edges which do not have subtype, reuse or category inheritance use but do have some other attribute (one of framework, generic, marker or constants).
	`numExplainedII/ numExplicitII`

`numCategoryExplII`
 	
`perCategoryExplII`
	Number of II edges which do have category use and do not have any other types of inheritance usage.
	`numCategoryExplII/ numExplicitII`

`numUnexplainedII`
 	
`perUnexplainedII`
	Number of II edges which do have an inheritance relationship, but which do not have any of the inheritance attributes defined.
	`numUnexplainedII/ numExplicitII`

Table 5.3: Interface Interface Metrics

5.4 Correspondence Between the Study Metrics and Article Metrics

The correspondence between the inheritance metrics used in the study and the abbreviations used in the published article are shown in table 5.4

Figures in article	Metric name in the article	Corresponding study metric
Fig. 10 and 11 CC Down-calls	Down-call proportion	`perCCDC`

Fig. 12 and 15
CC Usages
	INO
	EX-ST
	ST
		`perCCUsedOnlyInRe`
		`perCCExReuseNoSubtype`
		`perCCSubtype`

Fig. 13 and 16
CI usages
	UNK
	ORG
	SUS
	ST
		`perUnexplainedCI`
		`perCategoryExplCI`
		`perExplainedCI`
		`perOnlyCISubtype`

Fig. 14
II usages
	UNK
	ORG
	SUS
	RE-ST
	ST
		`perUnexplainedII`
		`perCategoryExplII`
		`perExplainedII`
		`perOnlyIIReuse`
		`perOnlyIISubtype`

Fig. 17
Other CC Usages
	UNK
	ORG
	SUP
		`perCCUnknown`
		`perCCUnexplCategory`
		`perCCUnexplSuper`

Table 5.4: Correspondence between metric names used in the article and the metrics used in the study.

Chapter 6

Replication Study

The main objective of our study is to validate the results of the original study by means of a replication. We attempt to replicate the WhaPDJI study with one important difference: we are using Java source code as input and the original study uses byte code. The main reason for choosing the source code as input is out programming language Rascal. Rascal can analyse Java source code but not the byte code. Moreover, we believe that the source code better reflects the intentions of the programmer than the byte code.

Except for this major difference, we tried to carry out the study in the same manner as in the original study. For example, we use WhaPDJI inheritance model and the metrics as is. Our study has its own limitations, and we discuss these limitations in detail in chapter 9. However, we also would like to mention here that our limitation about the content of our input is a major one and it will definitely have impact on our results. As mentioned already, the source code of the Corpus projects is analysed in our study. For almost two thirds of the projects we use different versions than those of the original study. Moreover, during the analysis we observed that the content of the source code and byte code distributions can be very different from each other. These two facts should always be taken into account when evaluating our results.

In this chapter, we explain the replication study in detail. Research questions are introduced first, followed by the artefacts we use. An explanation of the study set-up, especially how it differs from the original study ends this chapter.

6.1 Research Questions

The research questions for the replication are based on the results of the original study which were discussed in section 3.5. For all of the four questions, we look if our analysis produce comparable results. We will ask the following question for the four results which are reported by Tempero et al.:

Research Question 1: How do our results differ from the original study in down-calls?

Research Question 2: How do our results differ from the original study in subtype inheritance usage?

Research Question 3: How do our results differ from the original study in external and internal reuse?

Research Question 4: How do our results differ from the original study in other uses of inheritance?

6.2 Artefacts

In addition to the original article [TYN13] and the website of the WhaPDJI study [TYN08], we used the following artefacts when conducting our research:

Analysed projects We are using the compiled version of Qualitas Corpus, namely Qualitas.class Corpus [TMVB13b] as input to our analysis. We want to emphasize that we use the Java source code. Although the Qualitas.class Corpus is compiled, we still use the source code which also comes with this source code for analysis. The reason for this choice and its consequences are discussed in section 6.3.

In general, Java projects have dependencies on the external party libraries, which are available as java archive files (jars). For a project to successfully compile, such dependencies need to be found and resolved. This can be a time consuming task, especially if we have to deal with tens of projects. Terra et al. made a study [TMVB13b] and produced compiled version of Qualitas Corpus: Qualitas.class Corpus. Qualitas.class Corpus is available from the website [TMVB13a]. The consequences of this choice are discussed in section 6.3.3.

Analysis program The Java source code analysis program is written in Rascal. Analysis program is explained in a separate chapter (chapter 7) in this thesis.

Communication with the authors of the WhaPDJI study During the replication study some questions have arisen about the original study. We have communicated with the authors via e-mail, and our final results are also dependent on their answers as well as the other artefacts. Especially, the correspondence between the metrics from the Inheritance Use Website [TYN08] and the results in the article [TAD+10] have become clearer after mailing with the authors. The e-mails [Tem14] are available on request.

6.3 Differences in the Study Set-up

6.3.1 Source Code versus Byte Code

The biggest difference between the WhaPDJI and replication study is about the Java programs analysed. Tempero et al. used the byte code of Java open source systems, while we do our analysis on Java source code. The fact that the byte code does not always directly map on to source code, although in very rare cases, may cause some differences. This is discussed in detail in the original article, in section 4.3, Analysis Challenges. We do not think that this will cause a big difference between their and our results, because it happens very rarely.

An interesting difference between the byte and source codes arises when generics are used. The Java compiler applies type erasure to generic types and it also translates Java varargs to arrays. In the byte code, this *pre-processing* is already applied. In the source code, however, a different approach should be used when searching for the type information of generics. This approach presented a challenge in our study which is described in detail in the chapter 8.

Until this point, we've been talking about the differences between two different representations of the same code, i.e. differences between a Java class file (for example A.class) and a Java source file (A.java) of the same Java class A. There may also be some differences between what is included in a Java project (which set of classes and interfaces) in the byte code distribution and in the source code distribution. This potential difference may cause significant differences on the results and is discussed in detail in the following subsection.

6.3.2 Differences Between the Content of the Byte Code and the Source Code:

For each Java project included in Qualitas Corpus, two different distributions exist: binary and source. Authors explain in detail how they have obtained the byte and source distributions in ”Criteria for Inclusion in Qualitas Corpus” section in the web site of Qualitas Corpus [TAD+08]. They made the decision to take both distributions separately, and made the assumption that binary form would contain the compiled version of the source code. This assumption, is not validated, however. It may very well be possible that different sets of classes or interfaces are included in binary and source forms. It is highly probable that many classes and interfaces would be included in both of the forms, however, there may be differences between the two. Our experience has shown a difference and this is explained in detail in chapter 11.

We expect this to have impact on the results, since the set of analysed types will be different in the original study and in the replication study.

6.3.3 Version Differences

The byte code projects analysed in the original study are taken from the Qualitas Corpus [TAD+10]. We needed the source code of the same projects. These are available in the Qualitas Corpus. As explained earlier in this chapter, acquiring the source code is not always enough for a successful compilation of the code.

The original study used the 20101126 release of the Qualitas Corpus and that covers in total 93 projects. To be able to keep our input projects as close as possible to the original study, we first wanted to use the same versions of the projects analysed in the original study. Because of the time limitations, however, we decided to use the versions in the Qualitas.class Corpus. 66 projects have the same versions of the original study, while 27 are different.

Moreover, three projects from the Qualitas.class Corpus did not compile properly and we had to download the source code from internet. Three projects could not be analysed because of errors. In total we analysed 90 projects, 65 same version, 25 different version.

The list of projects (with the versions) we use in our analysis can be found in appendix A. Detailed information about the version differences in the analysed projects can be found in appendix B.

The different versions of a project will contain different code. This will inevitably introduce some differences in the results. As mentioned earlier, our decision was forced by time limitations and we report this decision as a limitation of the replication study in chapter 9.

Chapter 7

Implementation

In this chapter we explain how we implemented our Java source code analysis program. We used the meta-programming language Rascal. Rascal has many features for analysing Java source code. In Rascal, it is straightforward to create M3 model and the abstract syntax trees (ASTs) for a given project. Our whole analysis is made by using the information stored in the M3 model and by visiting the project ASTs. In this section, we will summarize our Rascal implementation for various types of inheritance relationships which are defined in chapter 4. The Rascal code of our analysis program is available in a git repository: [Ayt14c]. Moreover a large part of the source code is also appended to this document (C)

7.1 Internal Reuse Analysis

We do the internal reuse analysis only using the information stored in the M3 model. For each class in the project, we first retrieve all methods and fields defined in all ancestor classes of the investigated class (let's call it ancestorMembers). Then we retrieve the methods and fields which are defined in the investigated class (declaredInClassMembers). After this, we collect the methods which are invoked and fields which are accessed from within the investigated class (accessedMembers). For each item in accessedMembers set, we look if it is an element of the ancestorMembersSet and not an element of declaredInClassMembersSet. If an item satisfies these two criteria, we mark that class and its ancestor (of which a member was accessed) with internal reuse attribute.

The Rascal source code for internal reuse analysis is included in the appendix section C.1. The Rascal method `getInternalReuseCases` is the starting point.

7.2 External Reuse Analysis

For each class in the project, we build the class ASTs and retrieve all the field accesses and method calls in the AST which are issued on an expression (we can call this receiving expression). If the member (field or method) accessed is defined in a different type than the receiving expression, the pair <type of the receiving expression,type of the accessed member> gets the external reuse attribute. Let us call this pair <C,P> for convenience.

If <C,P> pair has explicit attribute (in other words, if the inheritance relationship between C and P is defined in the code), then we are done. However, if <C,P> pair is not explicit, then we are required to find the immediate parent and all other ancestors of C, up until type P in the same inheritance branch as C and P. Because of the definition in the WhaPDJI article, all pairs in this inheritance branch (between C and P) are going to have external reuse attribute.

If type P is not user defined, i. e., it is defined out of the project under investigation, we only mark C and its immediate parent as having external reuse. This is a limitation of our study and also listed in chapter 9 of this document.

Finding all the types between C and P is an interesting problem and is explained further in section 8.2.

The Rascal source code for external reuse analysis can be found in the appendix section C.2. The method `getExternalReuseCases` starts the analysis.

7.3 Subtype Analysis

As we have discussed in the definitions chapter (chapter 4), subtype usage can occur in many types of constructs: in an assignment, in a method definition (with a return statement), with casting, when passing parameter(s) to a method or constructor call, in an enhanced for loop, or in a ternary operation. We also visit the ASTs of each class in the project and investigate the nodes of the AST which involve these constructs. In all of the mentioned constructs we search if a child type is supplied when a parent type is expected. The analysis for subtype becomes challenging when analysing generic types with type parameters. This challenge is explained in chapter 8.

Just like in the external reuse analysis, in the subtype, the indirect usage can take place. For example, if a subtype usage has been seen between a grand child type and a parent type, both the relations <grand child,child> and <child,parent> is going to be marked with subtype attribute.

There are two cases of subtype analysis for which further explanation is necessary: This changing type analysis and down-casting.

Subtype analysis covers several cases and the type analysis of generics are also made here. The code for subtype analysis is in the appendix section C.3. `getSubtypeCases` is the method that starts the subtype analysis.

7.3.1 This Changing Type Analysis

This changing type concept was defined in section 4.7.3. The analysis is made in two steps. In the first step we search for the `this` references in the code (in initializers, methods or constructors) of each class, again by constructing and visiting the ASTs in Rascal. A proper candidate should not only have `this` reference, but if `this` reference is seen in a method, the class which contains the candidate method, should not override that method. We set up a list from such candidates. An item of this list contains descendant ascendant pair and the piece of code that contains the `this` reference.

The second stage uses the this candidate list as input and finds out the actual occurrences of `this` changing type:

If this reference was found in the initializer of the class, then we check if the object gets instantiated in the project,

If the reference was found in a method (or a constructor), and the method is called via a receiver expression, we check if the expression is of descendant type in the candidate list. In this manner, we make sure that the method was issued on one of the descendant types in the candidate list.

The Rascal module which does the this changing type analysis can be found in the appendix section C.4. The method `getThisChangingTypeCandidates` retrieves the candidates, the method `getThisChangingTypeOccurrences` uses the candidate list to find the occurrences.

7.3.2 Down-casting

Definition of down-casting was given in section 4.7.1. A valid question about subtype analysis is whether down-casting can be categorized as subtype usage. In the WhaPDJI article, casting was listed as one of the subtype usages, and up-casting and down-casting were explained as two types of casting. Nowhere in the study or in the article could we find an exception defined for down-casting which could exclude down-casting from the subtype analysis.

For this reason, we have chosen to include down-casting cases as contributing to subtype usage. In Java code, a down-cast from an object type to a user defined type can be seen frequently. For example, the following code segment in class N:

```
public class P { }
public class C extends P { }

public class GC extends C { }
public class N {
    public void n() {
        List list = new Vector();
        list.add(new GC());
        GC aGC = (GC)list.get(0); // get() returns Object
    }
}
```

which is frequently used when retrieving an object from a collection. We identify the down-cast (an object is cast to GC), and mark the relation between GC and Object as subtype. Because it is an indirect usage (GC is not direct child of Object), all the relations <GC,C> <C,P> and <P,Object> get the subtype attribute.

Since this down-cast from object to a user defined type can be seen frequently, this decision may increase the number of subtype usages.

7.4 Down-call analysis

Let us use a similar example from the definition section (4.8) to explain our implementation.

```
public class P {
    { q(); }

    void p() {
        q();
    }
    void q() {
    }
}

public class Q extends P {
    void q() {
    }
}

public class D {
    void d() {
        Q aQ = new Q();
        aQ.p();      // when p() is executed,
    }                //  Q#q() is called instead of P#q()
}
```

For the down-call analysis, we first collect all the overridden methods in the project in pairs. In the example, the `P#q()` method is overridden by the `Q#q()` method. We then find all the methods in the ascending class (P) and check if there is a method call to the overridden method from any of P's methods. If this is so, the pair <Q,P> is a candidate for down-call. It is also necessary to check that issuing method in the ascending class (method p() of P) should not be overridden in the child class Q. If it is not overridden we mark this pair as down-call.

Another down-call usage can occur in the class initializer. The method q() is called in the class initializer of P. When an object of type Q is instantiated, the class initializer of its parent class is called automatically. When this code executes, the q() of class Q will be called, instead of q() of class P, which again indicates a down-call. Here, we do not need to check any overriding, because class initializers can not be overridden in Java.

As we have explained in the definition of down-call, the question remains if the programmer really attempts to issue the down-call. In other words, do we see a method call on a Q object (in our example aQ.p()) anywhere in the code? The author (E. Tempero) indicated in his e-mail [Tem14] that they do not look for such a method issue. We have chosen also not to do so to be able to keep our study as close to the original study as possible.

For down-call implementation, some information is taken from the M3, however we also visit the ASTs to detect the method calls made from the ascendant types.

For the implementation of down-call analysis in Rascal please see C.5. Note that we implemented both the analysis of down-call candidates (in the original work, these are classified as down-call cases, which we also adhere to) and the analysis of down-call occurrences (namely, the receiver requirement we mentioned above). We do the analysis, but we do not do anything with the results of the down-call occurrences at the moment. We explain our suggestion for down-call analysis in the future work chapter 13, but here it will be sufficient to say that we have our analysis code ready for down-call occurrences, and this can be used in a future study.

7.5 Analysis of Other Uses of Inheritance

In this section we briefly explain how we implemented the other uses of inheritance.

7.5.1 Category

When a subtype relationship exists between a descendant type (let us call it C) and an ascendant type (P), for the siblings of the descendant type (S), we can suspect a category usage. If we can find no other relationship usage between S and P, we mark this pair as having category attribute. Category analysis uses the subtype analysis results. For each pair in the subtype results (say <C,P>) we look for the siblings of C by using information in M3. Let us say that we found a sibling (S) and we did not find any other inheritance usage between S and P so far. In this case the pair say <S,P> gets the category attribute.

Rascal code for category is included in the appendix section C.6.1.

7.5.2 Constant

For constant analysis, we collect all the types which only contain constants, in other words, static and final fields. For interfaces, all fields are defined by default as static and final; for classes, we check these qualifiers. For each such type, we also check if their ascendants (if any) also contain only constants. If this is so, we mark these pairs with constant attribute. For constants, information in the M3 model is sufficient, we did not have to deal with ASTs.

Please see the appendix section C.6.2 for Rascal implementation.

7.5.3 Framework

Framework implementation is straightforward and is also made using only information in M3. We collect all the non-system types first and then find their direct descendants (let's call these FDD1). Then we look for the direct descendants of FDD's themselves (FDD2). The ascendant-descendant pairs from FDD1 and FDD2 respectively get framework attribute.

Rascal implementation for framework is included in the appendix C.6.3.

7.5.4 Generic

The generic usage was defined in 4.9.4. For convenience, we list the necessary conditions for generic usage here again:

1. S is parent of R. (i.e. S is direct ascendant of R.)

2. R has at least one more parent, say, T.

3. There is an explicit cast from java.lang.Object to S.
4. There is a subtype relationship between R and java.lang.Object

We want to mention again that we did not implement the fourth condition (subtype relationship between R and object) because of time limitation. The third condition we search for by visiting the ASTs and looking for cast statements specifically. When we find a cast from object to a type (let us call it S), then we look for a direct descendant of S (let's call it R) which has at least one more parent (in the example T). We again use the information in M3 when searching for direct descendant of S and the other parent.

Generic Rascal implementation is included in C.6.4.

7.5.5 Marker

Marker implementation looks like the constant implementation with one difference. The ascendant in this case should be an interface and it should not have anything defined in it (let's call one such interface I). Classes that implement this interface (let's call one C) or other interfaces which extends from it (say, I2) will have marker usage. Thus the pairs <C,I>and <I2,I>will get the marker attribute.

For Rascal implementation of Marker, see the appendix section C.6.5.

7.5.6 Super

To find the super relations, we first get all the constructors of the project. For each constructor we build an AST and look if there is a super() call to the constructor of immediate parent. If this is the case, then the class of the child constructor and the immediate parent get the super attribute.

Super Rascal implementation can be found in the appendix section C.6.6.

7.6 Inheritance Logs

At the end of our analysis, as expected, we report all the inheritance pairs which involves some type of inheritance usage. In addition to this, the analysis program also produces logs which report the occurrences of these usages. We do a replication study ourselves, but we also would like our study to be replicable. Logs which list the details of inheritance usage may be useful if someone else wants to understand or replicate our study.

To be able to better explain the contents of the logs, let us consider the following example:

```
public class P {
    void p() {
    }
}
public class Q extends P { }

public class E {
    void e() {
        Q aQ = new Q();
        aQ.p();          // external reuse via method call
    }
}
```

Our external reuse log will contain the following entry for this code segment:

```
<< |java+class:///edu/uva/analysis/samples/Q|,|java+class:///edu/uva/analysis/samples/P| >,
   210,
   |project://VerySmallProject/src/edu/uva/analysis/samples/E.java|(101,6,<6,8>,<6,14>),
   |java+method:///edu/uva/analysis/samples/P/p()|   >
```

As we can see, the location of the occurrence (the method call aQ.p()) happens on a certain line of e() method of class E. This location is given in the source code reference:

```
|project://VerySmallProject/src/edu/uva/analysis/samples/E.java|(101,6,<6,8>,<6,14>),
```

If this entry is listed in the Rascal console, we can click on it, and Rascal will automatically open up the class E in Eclipse editor and highlight the referenced line. We find this Rascal feature very useful for later studies, because every inheritance attribute we give has associated occurrences with it (the location in the source code)

The first two entries in the external reuse log are the descendant ascendant pair. Third entry (in our example) 210 is the type of the external reuse, in this case "External Reuse Direct". We have already discussed the fourth entry, the source code location of external reuse. The last entry is the referenced method in the external reuse; in this example method `p()` of class P.

We have logs for the following types of inheritance usage: internal reuse, external reuse, subtype, down-call, category, generic, super and this changing type. For the constant, framework and marker attributes, we found logging not necessary, because we can not talk about a reference location for these attributes.

Different inheritance usage attributes will need different log structure. We found one example about external reuse sufficient here. The structures for each log and the different types of inheritance usage can be found in our Rascal code.

The inheritance logs for each project which are produced by our analysis programs can also be found in a git repository: [Ayt14b].

Chapter 8

Challenges of the Replication Study

Our study is based on the original study in almost all aspects. Therefore the challenges they have faced 3.4 should also be taken into account for our study: the third-party libraries are not analysed, and therefore the goal of some inheritance relationships can not be determined. The definitions of two uses of inheritance (framework 4.9.3 and generic 4.9.4) are based on heuristics.

Here we have one less challenge to deal with than the authors, the one about compilation: the source code does not always map correctly to byte code. Since we are analysing the source code and not the byte code, we did not face with this challenge.

Our major challenge was the analysis of Java generics. In the byte code, Java generic types are already translated to non-generic types, whereas in the source code this processing should still take place.

In this chapter we first talk about our analysis challenges, namely the type analysis of generics and finding the immediate ascendant of a given type.

8.1 Generics

To be able to explain the analysis of generics in Java, we should first give some information about generics in Java and type erasure. We have used the book [SB08] and the website [Lan13] when explaining Java generics and type erasure in this section. After that, we will explain how we dealt with type analysis of generics in our source code analysis.

8.1.1 Generics in Java

Generics in Java is a powerful feature which is introduced with Java 5. Generic types in Java are mostly used with Java collections or arrays to indicate which type of objects can be stored in a collection. A lot can be said about generics. Our intention here is not to explain the Java generics elaborately, we only explain the points which are necessary for understanding our study.

The following example illustrates the use of generics:

```
public class P { }

public class C extends P {  }

public class S extends P {  }

import java.util.ArrayList;

public class SubArrayList <T extends P> extends ArrayList <T> {
        T aT;
    T getAT() {
        return aT;
    }
}

public class N {
    void genericRun() {
            SubArrayList <P> aList = new SubArrayList <P> ();
        aList.add(new C());
        aList.add(new S());
        P aP = aList.get(0);
    }
}
```

In the class definition

`public class SubArrayList <T extends P> extends ArrayList <T> { }`

`<T extends P>` is the *type parameter* for class `SubArrayList`. `T` is called the *type variable*. `SubArrayList <T extends P>` is called a *generic type*. When a parametrized type will be instantiated, *type arguments* should be supplied. In our example, the statement:

`SubArrayList <P> aList = new SubArrayList <P> ();`

contains the type argument `<P>`. The type of the variable `aList`, `SubArrayList <P>`, is a parametrized type.

When instantiating a parametrized type, the type arguments on both sides should be the same for the same type (`SubArrayList`) . For example, the following is not allowed:

`SubArrayList <P> aList = new SubArrayList <C> ();`

In our analysis, we assume that the code compiles successfully, therefore we do not need to check if the type parameter supplied is suitable for the corresponding type variable.

8.1.2 Type Erasure

During compilation, Java compiler elides type parameters and type arguments, as explained in [Lan13]. This mechanism is called *type erasure*. Type erasure is already applied in the byte code. It is useful to list the differences between the source and byte codes here, since the authors and we are actually analysing different code when it comes to generic types. We will continue to use our example from 8.1.1. After the type erasure the code will look like this:

```
public class P { }
public class C extends P {  }
public class S extends P {  }

import java.util.ArrayList;
public class SubArrayList            extends ArrayList     {
    P aT;
    P getAT() {
        return aT;
    }
}

public class N {

    void genericRun() {
        SubArrayList      aList = new SubArrayList   ();
        aList.add(new C());
        aList.add(new S());
        P aP = (P)aList.get(0);
  }
}
```

The following changes are applied to the code:

- type parameters are deleted from the class definition and are replaced by their leftmost bound (in our case P):

```
public class SubArrayList             extends ArrayList      {
    P aT;
    P getAT() {
        return aT;
    }
}
```

- type arguments are deleted from the assignment statement

```
    SubArrayList     aList = new SubArrayList ();
```

- an explicit cast is inserted into the code when retrieving an element from the list:

```
    P aP = (P)aList.get(0);
```

8.1.3 Type Analysis of Generics

As mentioned earlier, we are using Rascal meta programming language for analysis. When we analyse a Java statement, we can obtain the type of an expression easily in most cases. In one particular case, however, we need to retrieve the type variables and type arguments and couple them to each other. This happens during the analysis of subtyping with parameter passing.

We will extend our example with an additional method to explain this case in detail:

```
public class P { }
public class C extends P {  }
public class S extends P {  }

import java.util.ArrayList;
public class SubArrayList <T extends P> extends ArrayList <T> {
    T aT;
    T getAT() {
        return aT;
    }

    void useT(T aT) {
    }
}
public class N {
    void genericRun() {
        SubArrayList <P> aList = new SubArrayList <P> ();
        aList.add(new C());
        aList.useT(new C());
    }
}
```

When we call the method `useT()` in `genericRun()` method of class N, we pass it a parameter of object type C. The variable `aList` is instantiated with type argument P, and therefore in this context, the method call `aList.useT(new C())` will expect an argument of type P. We mark this case as subtype usage because parameter of type C is supplied when parameter of type P is expected.

This analysis in Rascal presented a challenge because the method declaration of `useT(T aT)` in Rascal refers to the type variable T. Here, it is necessary to find out with which type arguments the variable `aList` is instantiated in the first instance. In our analysis we couple the type arguments with type variables one by one and find out that the method `useT()` which is issued on variable `aList` will actually accept parameter of type P.

Type parameters can also be used in a nested manner, as shown in the short example below. We include these occurrences in our analysis as well:

```
ArrayListParent <NestedParent<P>> nestedListParent;
```

This presents the most important challenge we faced during the code analysis.

We are confident that we analyse most of the generic types properly, however we can not claim that we do cover all the cases without any exceptions. This fact introduces a limitation , and listed again in chapter 9.

The implementation of the type analysis can be found in appendix section C.7. The method `updateTypesWithGenerics` finds the references between the type variables and the type arguments for the declared parameters of a method.

8.2 Finding the Ancestors up to a Given Type

As explained earlier in subtype and external reuse analysis, if the usage for these attributes occurs indirectly, then the usage attributes are propagated up in the inheritance hierarchy until the indirect ancestor has been found.

Let us consider the following example:

```
public class P {
    void p() {  }
}

public class C extends P { }

public class GC extends C { }

public class N {
    public void runIt() {
        GC aGC = new GC();
        aGC.p();
    }
}
```

The indirect external usage between GC and P causes that, both <GC,C> and <C,P> pairs get the external reuse attribute. To be able to find all the types between GC and P in the inheritance hierarchy, we implemented an algorithm.

In our implementation, GC is named descendant type and P is the given ascendant type. If both of ascendant and descendant types are classes, then we search the in between types only among the classes. The same also holds for the interfaces as well; if both descendant and ascendant types are interfaces we only look for interfaces as the types in between. If the descendant type is a class, and the ascendant type is an interface, the search can go in both ways, namely, we may have classes and interfaces as in between types. In this case, we search for the immediate parent of the given class first among the interfaces, and if we can not find an interface as immediate parent, we search the immediate parent among the classes.

Chapter 9

Limitations

In this chapter we report the list of limitations, together with the impact they may have and the reason for having the limitation. The items are listed according to their importance. The difference between the input to analysis in the original and replication studies is our biggest limitation and it is going to impact the results: We analyse different versions for almost one third of the projects. More important than this, the source code distribution contents are in many cases different from byte code distribution contents.

1. **Different versions of 25 projects are analysed:** For 25 projects, we have analysed different versions than in the original study, since we have chosen to use Qualitas.class Corpus [TMVB13b]. The version differences are listed in appendix B (reason: time limitation).

2. **The content of the project source and byte codes are different from each other:** The authors of the WhaPDJI study retrieved the byte and source codes separately from the project websites, and they assumed that the byte code will contain the compiled version of the source code. For many projects we have analysed, we have seen that this was not true (reason: limitation introduced by the original study).

3. **Limited analysis of non-system methods during subtype analysis via parameter passing:** If a method is not defined in the project, we can not obtain the types of the method parameters from Rascal. For this case, we introduced a very simple heuristic. We cover only the methods which have interface or classes as parameters. The non-system methods which have primitives, generic types or varargs are not analysed for subtype via parameter passing. This limitation may result in less subtype pairs than the original study (reason: both technical and time limitations).

4. **Limited addition of pairs in the inheritance hierarchy when analysing indirect external reuse via non-system methods:** When there is an indirect call to a non-system method, only the immediate parent of the descendant type is taken into account. For system methods, however, we add all the pairs in the inheritance hierarchy up to the defining type of externally reused method. The reason of this limitation is that we do not know the type which defines the method for non-system methods. Because of this limitation, it is likely that fewer external reuse pairs will be reported (reason: technical limitation).

5. **3 projects are not analysed:** Three projects, which are listed in appendix A could not be analysed because of errors.

6. **Internal reuse attribute is not analysed for class interface and interface interface pairs:** Because of this limitation, the number of unexplained class interface and interface interface pairs may be reported higher (reason: time limitation).

7. **Byte code and source code difference when analysing generic types:** As explained in the section 8.1, byte code and source code are very different for generic types. Some differences

may occur between the WhaPDJI study and our study because of this difference. We are confident that we analyse most of the cases correctly in our type analysis, but still can not claim that we cover the whole possible cases (reason: time limitation).

8. **Method calls with ternary operators as arguments:** When doing subtype analysis via parameter passing, we do not analyse the method calls that have ternary operators as arguments (reason: time limitation).

9. **Limitation about generic attribute:** Generic usage is one of the other inheritance cases and it is explained in subsection 4.9.4. As explained in that subsection in detail, we do not look for the last rule for the generic definition in the code, namely, the subtype relationship between descendant type and java.lang.Object. This may increase the number of generic cases we report, since we are less strict in our analysis (reason: time limitation).

10. **This changing type attribute is not analysed for class interface pairs:** This changing type attribute is part of subtype analysis and is defined in 4.7.3. We do this analysis for class class pairs, but not for class interface pairs. This may result in fewer subtype pairs reported in the class interface results (reason: time limitation).

11. **For interface-interface pairs category attribute is not analysed:** This may also cause fewer explained interface interface pairs and as a result more unexplained interface interface pairs (reason: time limitation).

12. **Some external reuse cases are also counted as internal reuse case:** When an external reuse occurs between a descendant and ascendant pair, and if this reuse happens not in the descendant itself, but in a type that has inheritance relationship with descendant, then this case is counted twice, as external reuse and internal reuse. However, because the metrics report internal reuse pairs which are not external reuse pairs, this error in our program does not affect the analysis results (reason: time limitation).

Chapter 10

Results of the Replication Study

The results of the replication study are presented in this chapter. For the most part, we explain our results which are related to research questions. At the end of this chapter we also list the results of our study and the original study next to each other for all study metrics. We would like to repeat here our two analysis limitations: we could do the analysis of all but three systems and we analyse different versions of 25 projects from total of 90.

We also include the results of the original study where needed for comparison reasons.

The Excel sheet in which the final results can be found is also in the git repository: [Ayt14a].

10.1 Down-call results

The median of number of down-call pairs we observed is 27 %. This is less than the median found in the original study, namely 34 %. Our results also vary between the projects and we also do not see any correlation between the size of the project and the number of down-call pairs.

For the projects that are mentioned about down-call usage in the original article, we have the following results: For the projects freecs and jasml, we did not observe any down-call pairs, just like the authors. freecs contains 61 CC pairs and jasml 22 in the source packages we analysed. For megamek (1264 pairs), the authors observed no down-call usage, while we did, about 11 %.

For the project ant (version 1.8.1) we observed about 29 %, and for freecol (version 0.10.3) 27 %. the authors report 28 % for the same version of ant and 35 % for freecol (version 0.9.4).

In figure 10.1, our down-call analysis results are shown and in figure 10.2 the results of the WhaPDJI study. It shows the proportion of down-call pairs to the total number of explicit user defined class-class pairs.

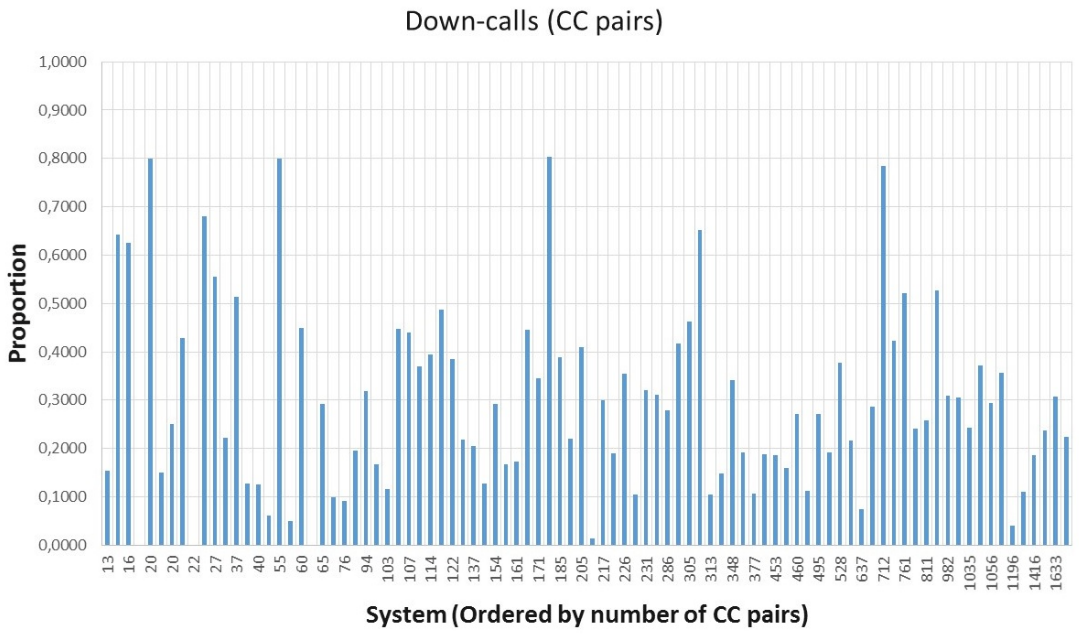

Figure 10.1: Replication study - proportion of down-call pairs

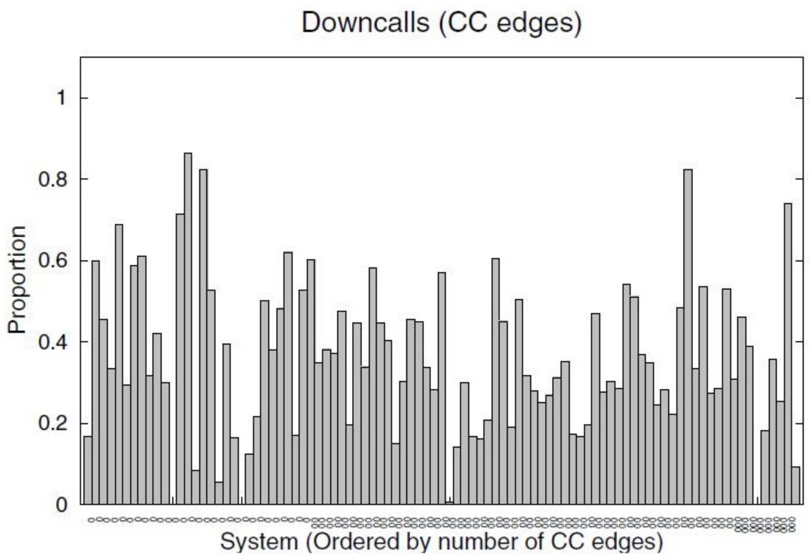

Figure 10.2: Original study - proportion of down-call pairs

We agree with the conclusion of authors about down-call which states that late-bound self-reference plays a significant role in the systems that are studied. We have observed less down-call cases (median %27) than theirs (% 34), but the usage we found is also significant (in our case, almost one fourth of the all CC pairs have down-calls.)

We have some arguments about the difference between the reported down-call results in both studies. We discuss these arguments in chapter 11.

10.2 Subtype Usage

The subtype usage can occur between all class class (CC), class interface (CI) and interface interface (II) pairs.

For class class pairs, we would like to emphasize one difference in the metrics here. The three usage metrics for CC pairs (subtype, external reuse - no subtype and only internal reuse) are perCCSubtype, perCCExreuseNoSubtype and perCCUsedOnlyInRe respectively. These percentages are the ratio of number of these three cases to the *used* pairs, and not to the *defined* pairs. The usage percentage (the number of pairs which have subtype, external reuse or internal reuse divided by the total defined CC pairs) is high for many projects. For the original study the median is 99 % and for the replication 88 %. When we talk about the subtype or reuse usage percentages for CC pairs, this detail should be taken in to consideration.

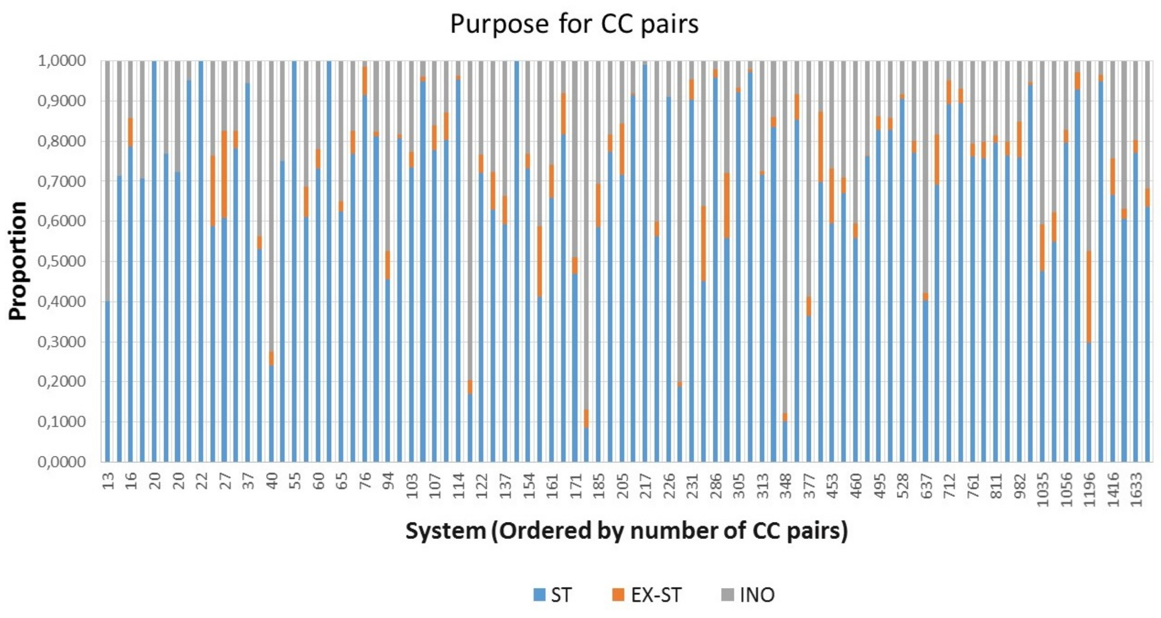

Figure 10.3: Replication study - Purpose for CC pairs ST: Subtype, EX-ST: External Reuse but not Subtype, INO: Internal Reuse Only

The results of replication study and the original study about CC usage are shown in figure 10.3 and 10.4 respectively. In both figures ST stands for subtype usage, EX-ST for external reuse but no subtype and INO for internal reuse only, i.e. pairs which have internal reuse but no subtype or external reuse.

The smallest subtype usage we see is the project displaytag-1.2 with 8,5 %, original study reports 15 % for the same version. The minimum subtype usage of the WhaPDJI study is seen in checkstyle project (version 5.1) with 11 %. We use version 5.6 and we found approximately the same subtype percentage for this project: 10 %. Both jasml (version 0.10 and 21 CC pairs) and megamek (version 0.35.18 and 1283 CC pairs) projects have subtype percentage of 100 % in the WhaPDJI study. We found for jasml (same version but 22 pairs) also 100 %. For megamek (same version and 1263 pairs), we observed 95 % subtype usage.

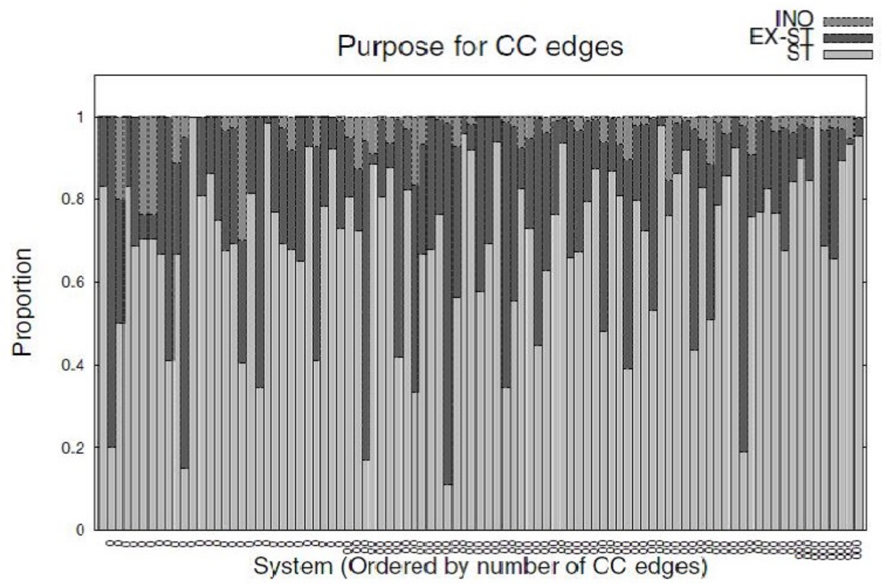

Figure 10.4: Original study - Purpose for CC pairs

The median for CC subtype usage in replication study is 76 %, which is the same as the original study (also 76 %).

Jre is the largest system analysed in the original study with 5735 CC pairs and about 95 % subtype usage. We could not analyse jre because of errors. The largest system we analysed is lucene (version 4.2.0, 2915 CC pairs), we found about 64 % subtype usage in this project. The WhaPDJI study analysed version 2.4.1 of lucene, with 446 CC pairs and found out 72 % of subtype usage for this project.

Also for class interface (CI) pairs, the subtype dominates the usage. Just like the authors, we also found that project fitjava has no CI pairs. Authors report 100 % of subtype usage for the following projects: nekohtml (8 CI pairs), jsXe (3 pairs) and joggplayer (5 pairs). We analyse the same version of all these three projects, however our number of CI pairs is different, 19, 13 and 21 pairs respectively. We found subtype percentages of 78 %, 85 % and 29 %.

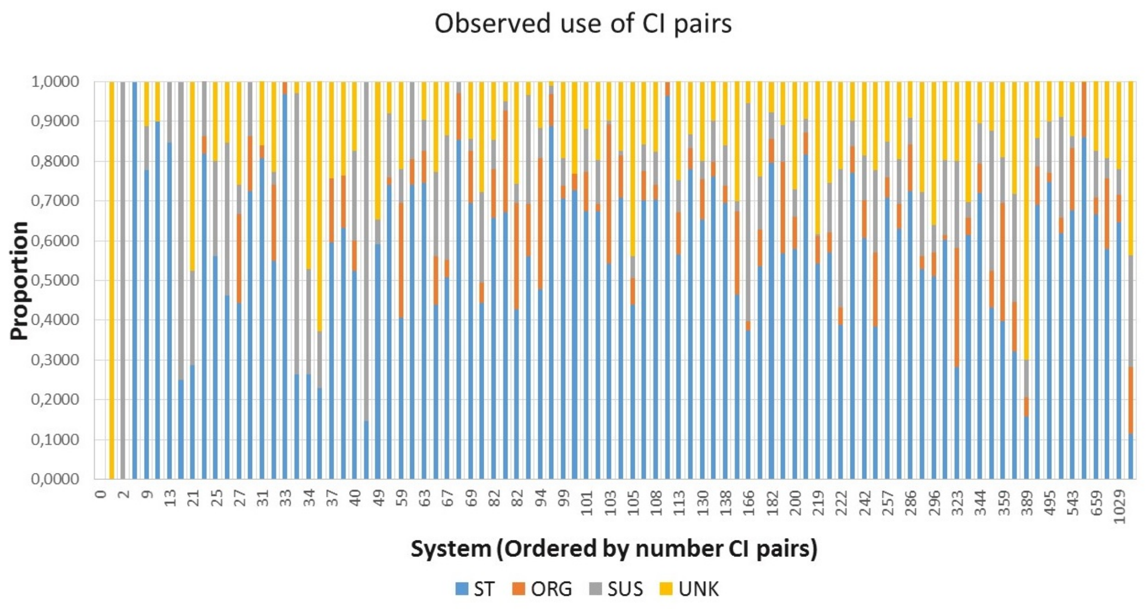

Figure 10.5: Replication study - Observed use of CI pairs

We did observe 100 % CI subtype usage only in one project, javacc (8 pairs). WhaPDJI study reports four projects which have subtype or suspected subtype usage, namely jasml, jmoney, jparse and javacc. We also found four such projects: jasml, jmoney, jparse and jsXe.

The authors report a median use of 69 % for subtype usage, and 85 % if they also add suspected subtype usage if all projects CI pairs are considered. In our study, we found median of 61 % for subtype usage and 75 % for subtype and suspected subtype together. The results are shown in figure 10.5 (for replication) and figure 10.6 (for original study). ST stands for subtype, SUS for suspected subtype use, ORG for organizational (category) and UNK for unknown purpose.

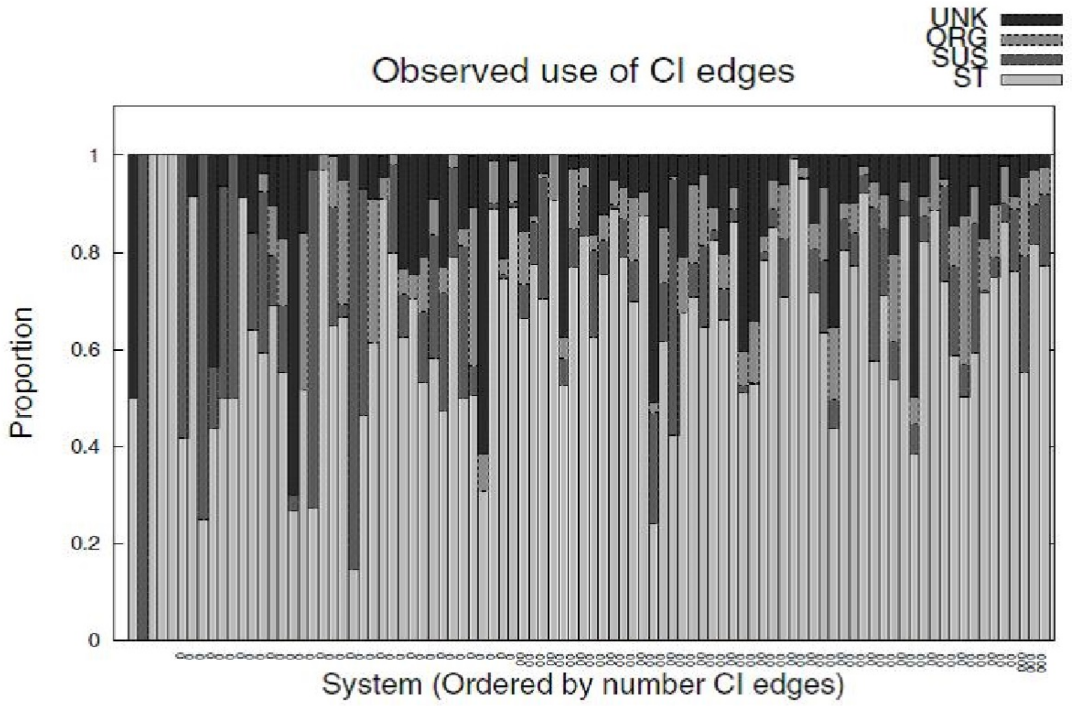

Figure 10.6: Original study - Observed use of CI pairs

WhaPDJI study reports for ant (version 1.8.1) the CI subtype only percentage of 63 % and together with suspected subtypes of 78 %. For freecol (0.9.4), these percentages are 83 % and 94 % respectively. For the same version of ant we found percentages of 38 % and 59 %. For freecol (version 0.10.3), our percentages are 43 % and 47 %.

Finally, we will explain our II (interface interface) results. There are many projects which do not have any II pairs. We found 23 such projects, just like the authors. Total number of projects with fewer than 10 pairs is also in both studies 51. We found 16 projects with 100 % subtype II use, the authors 13. The largest project among these 16 is picocontainer with 9 II pairs. In general, we found median of 75 % subtype usage and the authors 63 %. For ant (18 pairs) and freecol (3 pairs) (for the versions given above), they report 94 % and 67 % subtype usage, while we found 72 % (ant - 18 II pairs) and 100 % (freecol only 1 II pair). nakedobjects had the largest number of II pairs (349) for which we found 53 % of subtype use, whereas the authors report 66 %. Both of the studies use the same version of this project.

II use is shown in figure 10.7 for replication study and in figure 10.8 for the original study.

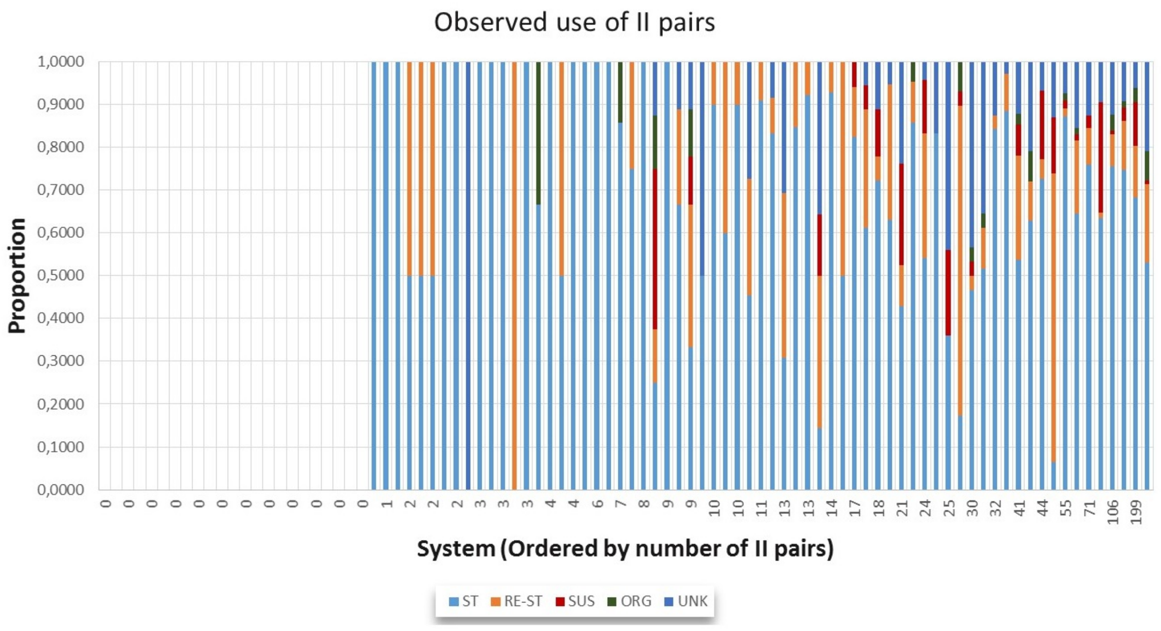

Figure 10.7: Replication study - Observed use of II pairs

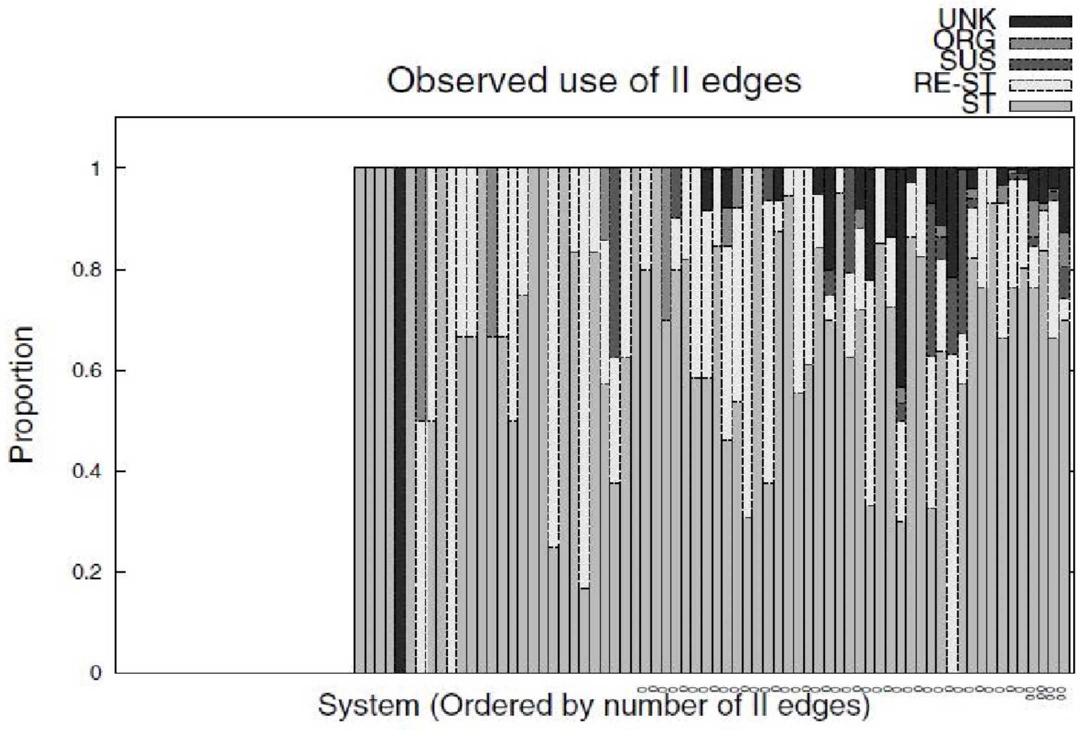

Figure 10.8: Original study - Observed use of II pairs

The authors conclude that "at least two thirds of all inheritance edges are used as subtypes in the program...". We also see the subtype usage is more than 60 % among the *used* pairs. But when we consider all pairs (not only the used ones) for CC pairs, we observe 64 % median subtype usage while authors report 72 %.

10.3 External and Internal Reuse

For the third research question (To what extent can inheritance be replaced by composition?), we should look at the pairs which have external or internal reuse, but not subtype use.

For external reuse, the authors report checkstyle (version 5.1, 193 CC pairs) with the largest proportion of external reuse (88 %). For the same project checkstyle (different version, 5.6 and 348 pairs) we found 2 % only. Median of external reuse in the WhaPDJI study is 22 %, whereas we report 4 % percent. This is a big difference and we explain possible reasons for this difference in chapter 11.

For internal reuse, the project with largest proportion of internal reuse was 30 % which was found in project jpf, version 1.0.2, 37 CC pairs. We analysed the 1.5.1 version of jpf (39 CC pairs) and found 44 % pairs with internal reuse attribute. The median for internal reuse is 2 % in the original study, while our median is 20 %.

The authors conclude that there is enough opportunity for replacing inheritance with composition, with 22 % external and 2 % internal reuse pairs. Our results also support this conclusion, with 4 % external and 20 % internal reuse pairs. However external and internal reuse percentages are very different from each other between the two studies, we report similar percentages for reuse in total.

10.4 Other Uses of Inheritance

We classified our results about other uses of inheritance according to the metrics given in the study as described in chapter 5. The authors include in their article a more detailed classification. There are metrics which measure the super and category use of CC pairs, however, there are no metrics which give detail about constant or marker attribute for CC edges.

Because of time limitations, we only report the results of the explained metrics and we do not go further in detail about certain attributes of inheritance like constant, framework or generic. To avoid misunderstanding, we would like to say that we analyse these attributes one by one, however, we give less details when we report our results.

In this section, we will explain the results we found according to the metrics mentioned before.

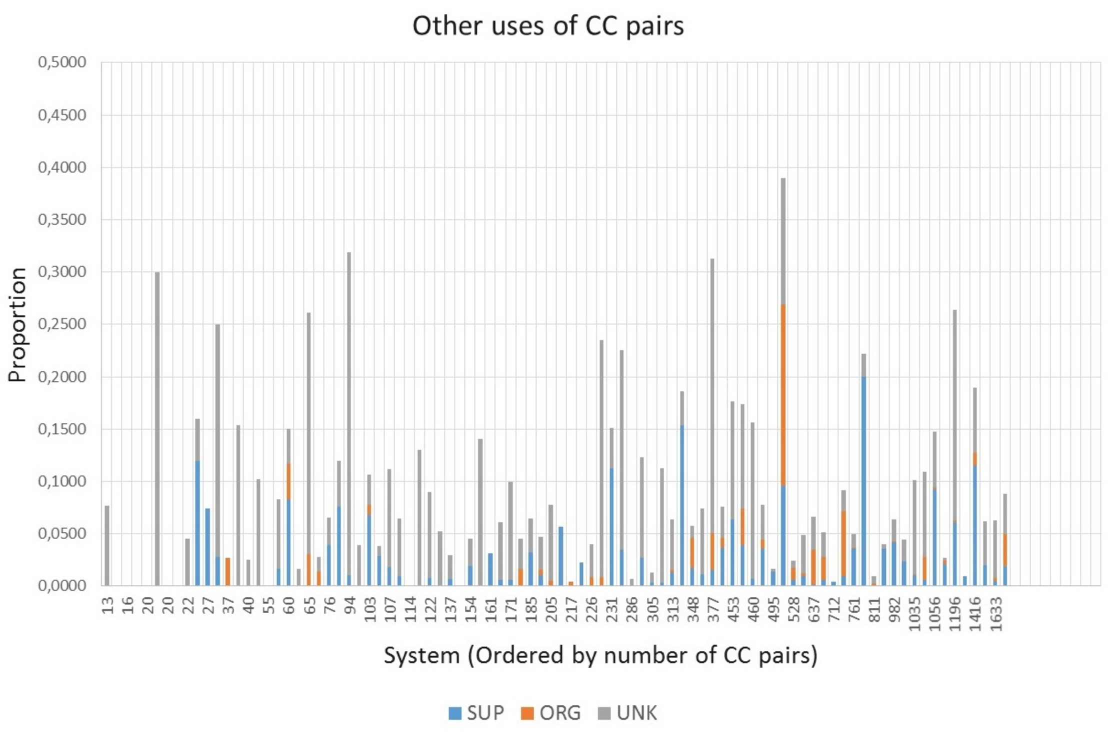

Figure 10.9: Replication study - Other uses of CC pairs

For class class (CC) pairs, we detected super attribute in many projects. About 60 projects have this attribute, although in small percentages. Median is about 4 %. jtopen (version 7.1 with 806 CC pairs) have the largest percentage (about 20 %). For category cases, about 36 projects have one or more these cases. The project with the largest percentage of category cases is fitlibrary (20110301 version, 498 edges): 17 % of all CC pairs. Median is 0, and the average is 0,7 %. Most of the CC pairs that have inheritance relationship could be classified in one of the usage categories. Our median for CC pairs with no purpose (unknown purpose) is about 3 %. This means that for 97 % of the CC pairs a reason for usage could be found. The unknown percentage is the highest in maven project (version 3.0.5, 94 CC pairs): 31 %. The medians for super, category and unknown cases in the original study are all 0 %, however, they also detect many projects with these attributes.

Class interface (CI) pairs have also other inheritance uses. The median for category use is about 7 %. Explained CI cases, which include framework, generic, marker or constant use has a median of 9 %. Finally we report median of 16 % for unexplained CI pairs. In the original study the medians for category, explained and unexplained CI cases are: 5 %, 7 % and 8 % respectively.

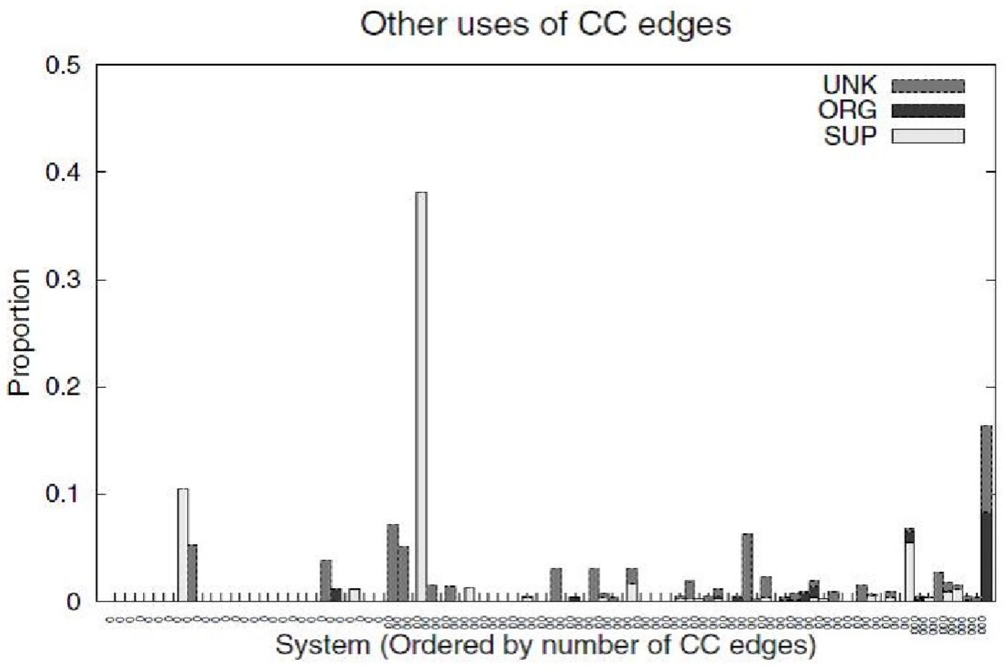

Figure 10.10: Original study - Other uses of CC pairs

The metrics about other inheritance uses of interface interface II pairs are (similar to CI pairs) category, explained and unexplained uses. Our medians for all these three metrics are 0, just like original study. We see other uses in various projects. For example, out of 69 projects which have II pairs, about 24 projects have at least one case of explained use and 16 category use. For 35 projects we could find some pairs without any known usage.

An important question here is how many percent of all pairs can be explained by one of the subtype, external reuse or internal reuse attributes. For CC pairs, authors report this as 95 % (= used CC pairs / total CC pairs, 38122/39973). We found 85 % (29547/34956). For all CC, CI and II pairs, they report about 87 % (58536/67529). In our case, this is 76 % (40376/53411).

Although our results are lower than those of the original study, our study also shows that vast majority (76 %) of all inheritance relationships can be explained by one of the subtype, external or internal reuse attributes. To conclude, we can say that other uses of inheritance occur, but in total, they are considerably less than subtype and reuse together.

10.5 Comparative Summary of Results

To summarize, we present the results for the study metrics in table 10.1. The explanation about each metric was given in chapter 5.

Metric	Replication median (%)	Original median (%).
`perCCUsed`	88	99
`perCCDC`	27	34
`perCCSubtype`	76	76
`perCCExreuseNoSubtype`	4	22
`perCCUsedOnlyInRe`	20	2
`perCCUnexplSuper`	1	0
`perCCUnexplCategory`	0	0
`perCCUnknown`	3	0
`perOnlyCISubtype`	61	69
`perExplainedCI`	9	7
`perCategoryExplCI`	6	5
`perUnexplainedCI`	16	8
`perIISubtype`	75	72
`perOnlyIIReuse`	9	17
`perExplainedII`	0	0
`perCategoryExplII`	0	0
`perUnexplainedII`	0	0

Table 10.1: Comparative Summary of Results

Chapter 11

Discussion

In this chapter we will discuss the results of the replication study.

Most important point about our results is that they are similar to the results of the original study, however, they are not the same. Furthermore, our results are close to the original results for some usages (subtype usage, reuse total and other uses of inheritance). On the other hand, our total usage (total of subtype, external reuse and internal reuse) is lower than of the WhaPDJI study (median of 88 versus 99 percent), just like the down-call median (27 versus 34 percent). Although the total reuse percentages are close to each other (23 vs. 24 percent), there is a big difference in individual reuse percentages: for external reuse 3 vs. 22 percent, for internal reuse 20 vs. 2 percent.

We believe that the major reason for these difference is the input to analysis. This will have affect on all types of inheritance which we discussed, and therefore should be taken into account for all results. Our input is the source code distributions of Qualitas Corpus projects: the content of the source code distribution is, for many cases, is significantly different than the byte code distribution. The number of explicitly defined class class relations for some example projects are shown in table 11.1. Moreover, for more than a third of the projects we analyse different versions than the original study. We also mentioned these important input differences in chapter 9.

Name of the project	Byte code	Source code
cjdbc-2.0.2	293	460
cayenne-3.0.1	940	1633
checkstyle-5.1	193	348
exoportal-v1.0.2	378	1050
htmlunit-2.8	458	637
jchempaint-3.0.1	357	459
jtopen-7.1	1347	806
openjms-0.7.7-beta-1	840	231
struts-2.2.1	536	1035
trove-2.1.0	126	14

Table 11.1: Number of external user defined class class pairs in byte and source code

In addition to differences in our input contents, our differences in interpretation of inheritance may also cause some divergence in our results. However, we have mailed with the author (E. Tempero) about three times to ask questions about the definitions and we are confident that we have a good understanding of the definitions by now.

For each research question in particular, we discuss the results and the possible reasons for differences in the following sections:

11.1 Down-call

For down-call research question, we agree that significant amount of class class pairs involve down-call usage, however, we report fewer cases of down-call (median of 27 %) than the original study (34 %.)

For down-call analysis results, we do not expect the method analysis limitation to contribute to the lower percentage we report. This is because the authors, just like us, analyse only system classes and system methods for down-call. We were curious about why we reported fewer cases. To find out the reason, we found a project which was small enough to manually inspect and also for which our down-call results differed considerably: project cobertura. We also mentioned this project when we explained our threats to validity. We did not observe any down-call cases for this project, while the authors report 5 of them. The version of the project and the number of CC pairs (17) are also the same in source and byte codes. These all made cobertura an ideal candidate for down-call comparison. We have made manual inspection in the source code of the project for all five reported cases. We will discuss here two cases to illustrate what the reason for this difference can be.

The first case is about cobertura class `GTToken` which we also discussed before:

```
public static class GTToken extends Token
{
  int realKind = JavaParser15Constants.GT;
}
```

We do not see any methods in this class. According to the definition of down-call the descendant class should override at least one method of the ascendant class (in this case Token). Because the authors analyse the byte code, we looked into the byte code if there was a method definition. Here is en excerpt:

```
  0: aload_0
  1: invokespecial #10 // Method
          //  net/sourceforge/cobertura/javancss/parser/java15/Token."<init>":()V
  4: aload_0
  5: bipush        126
  7: putfield      #12   // Field realKind:I
 10: return
```

It seems here that there is at least a call to `<init>` method of the ascendant `Token`. Even if this is so, we can not see an attempt from the programmer to issue a down-call (as defined in the original study) in the source code and therefore we think that this case should not be counted as down-call.

The second case we will discuss is the class `ClassData`. We have inspected this class and its ascendant `CoverageDataContainer`. We list here the part of the classes for which we suspected a down-call:

```
public class ClassData extends CoverageDataContainer
            implements Comparable<ClassData>, HasBeenInstrumented {
    public int getNumberOfCoveredBranches() {
        // ...
        number += (i.next()).getNumberOfCoveredBranches();
        // ...
    }
}

public abstract class CoverageDataContainer
        implements CoverageData, HasBeenInstrumented, Serializable {
    // ...
    public double getBranchCoverageRate()    {
        // ...
        CoverageData coverageContainer = iter.next();
        number += coverageContainer.getNumberOfValidBranches();
        numberCovered += coverageContainer.getNumberOfCoveredBranches();
        // ...
        }
    // ...
    public int getNumberOfCoveredBranches() {
            // ...
        coverageContainer.getNumberOfCoveredBranches();
        // ...
    }
    // ...
}
```

In the class `CoverageDataContainer`, the method call `getNumberOfCoveredBranches()` in the `getBranchCoverageRate` method is our candidate for down-call. However, this method is called on an object of type the `CoverageData`. For this reason, the `getNumberOfCoveredBranches` of `CoverageData` is going to be called. `CoverageData` is an interface, so the actual class of this method will be determined in run-time. We do not consider this as a down-call case, because in our interpretation the receiver of the down-call (in this case, `coverageContainer`) should have the type of the ascendant, i.e. `CoverageDataContainer`.

In his answer to our question, E. Tempero writes : " I'm fairly sure Case 3 (CoverageDatContainer case above) is an interaction between the conservative nature of the analysis (as referred to in the ECOOP paper) and a difficulty distinguishing self-calls in the byte code." He further explains the issue with an example, and this text is available in the e-mail communications [Tem14]. It seems that there is a difference in the interpretation of down-calls for this particular case. The definition in the original article does not discuss these kinds of cases. If we only read the article, we think that this type of cases should not be counted as down-call. For the final decision, we need more discussion with the authors. For this replication study we have chosen not to classify these type of cases as down-call.

Again, we can not prove that the 7 % difference between the down-call median of original (34 %) and replication (27 %) studies is the result of these two types of cases, but we think that this is the major reason for it.

11.2 Subtype

For research question two (about subtype usage), our results also agree with the results of the original study, but they are not the same. Subtype is the dominating inheritance use in both of the studies. Our results are the same if we look at the perCCSubtype metric (76 % median in both cases). This metric calculates the number of class class subtype pairs among the *used* pairs. If we look at the ratio of the subtype pairs among all defined CC pairs, we reach to different results, however. We have then a median of 64 % and the original study 72 %. We should repeat that we analyse different set of types here, but the other major reason for this difference, we think, is our limitation in method analysis. If we encounter a non-system method, we only apply a very limited heuristic for analysis of the parameters of the method. It is highly probable that we miss subtype cases that actually exist because of this limitation.

11.3 Reuse - External and Internal

The third research question is about if there are opportunities for replacing inheritance with composition. To be able to replace inheritance with composition, the authors look at the external and internal reuse cases for which no subtype usage was seen. Our results also show also significant opportunity (23 % total reuse) for such a replacement. The distribution of the total reuse between the internal and external is very different in our case. We report a much lower median of external reuse percentage than the authors (3 % vs. 22 %), and a much higher median for internal reuse (20 % vs. 2 %). This does not play a significant role for the answer of this research question, since the total reuse is the main factor which determines the replacement opportunities.

We still find it important to explain these differences, though.

Although our time limitation did not allow us to discuss this via e-mail with the authors, we again suspect an interpretation difference for external reuse. It is about the method calls or field accesses via a receiver, where the type of the receiver is the same of the class containing the method call. We will illustrate this case with an example:

```
public class P {
    void p() {  }
}

public class Q extends P {
    void q() {
        Q aQ = new Q();
        aQ.p();         // internal or external reuse ?
    }
}
```

According to the external reuse definition of the original article, we can not classify call `aP.p()` as external reuse: "An edge from type S (child) to T (parent) has the external reuse attribute if there is a class E that has no inheritance relationship with T (or S), it invokes a method m() or accesses a field f on an object declared to be of type S, and m() or f is a member (possibly inherited) of T." [TYN13]. We have chosen to classify this case as internal reuse, but we do not know if the authors also did so during their analysis.

Another possible reason is very similar to down-call case about class `GTToken` in cobertura which was discussed elaborately in section 11.1. In the logs of the original study, the pair <GTToken,Token> is also classified as external reuse (to be precise, direct external reuse method call.) In our analysis, we did not identify this as such and we manually inspected cobertura project to find out why. We searched for the project for this type and out of the references we could find, none issued a method call. Just like in the down-call, the byte code analysis may cause a false positive about the external reuse as well (`invokespecial` call).

Finally our analysis limitation about the non-system methods and the non-system fields may be playing a role here. When we determine an external reuse to such a method or a field, we mark the containing type and the immediate parent as having external-reuse attribute and we do not propagate the attribute all the way up in the inheritance hierarchy. This means that we include the first pair of the hierarchy only and we report possibly fewer cases of external reuse in this case.

For internal reuse, we think that our higher percentage is due to our lower percentage in external reuse. The study metrics report first the external reuse cases, and then the internal reuse cases which are not included in the external reuse cases. We include considerably fewer cases in the external reuse, so our internal reuse results are not excluded because of being external reuse.

11.4 Other Inheritance Cases

For the fourth research question, we also saw (just like the authors) that the other cases of inheritance existed, but they were not significant in percentage.

Although the other inheritance cases do not play a significant role in the inheritance usage, it is still useful to explain a few points. We report a median of 3 % for unknown CC pairs, while original study reports 0 %. We can conclude that some pairs which we can not place in any category (subtype, external reuse, etc...) were placed in some category by the original study. We see a similar case in CI cases. We report a median of 16 % of unexplained pairs, whereas the original study reports 8 %. Probably some pairs we do not report as subtype shift to the unknown category, because we report 61 % of subtype for CI pairs, whereas the WhaPDJI study 69 %.

Chapter 12

Threats to Validity

The most important threat to validity for our replication study is our input. There are differences in versions and in the content of analysed projects, as we discussed in chapter 9. Three projects we did not analyse at all, and for 25 projects we use different versions than the original study. Moreover, even for the projects for which we analyse the same versions, the Java types that are included in the byte code and source code are different from each other. For comparison reasons, we report the number of explicit user defined class class pairs in byte and source codes for some projects in table 11.1. The fact that we do not analyse the exact same content for many projects is a threat to validity for our replication. Moreover, we can not estimate the impact of this difference at all; we just know that there will be *some* difference.

One other threat to validity is the possible differences between our interpretation of inheritance usage definitions and of the original study. We started our study with the definitions given in the original article [TYN13]. As we progressed, we also found out more about definitions and metrics in the website of the original study [TYN08]. Even after that, we had some important questions about the definitions. We wrote e-mails to the authors [Tem14] and received answers from the first author, Ewan Tempero. We received clear answers for many of our questions. We used all this information in our implementation. For many definitions, we think that we have the same understanding as the authors. However, we can not guarantee that for each and every case without discussing every detail with the authors.

For a few cases, the authors could not give clear answers. As mentioned before, we have chosen a small project (cobertura) out of the Qualitas Corpus and compared our results with the original results. We could not observe any down-call cases in this project, while the authors reported five of them. We scanned the code for these cases manually and still could not see any down-calls. We e-mailed the authors with these findings and asked their opinion. For two cases, they did not have any explanation. For the other three cases, they suspected an error in their analysis, but could not be sure about it. There are several reasons why the author could not give clear answers: the analysis was made a while ago (three years ago), the tool chain they are using is long and complicated, he is missing some data files and hence he could not reproduce the results. For these cases, we can not guarantee the correct interpretation of the definitions, and therefore this forms a threat to the validity of our results.

Third threat to validity originates from the difference between the source code and the byte code. As explained earlier, source code looks very different from the byte code in some cases, for example, for the generic types. These differences may result in differences in analysis. The following example from cobertura is reported as down-call case in the original study, whereas we do not see down-call for this class in the source code:

Source code of class GTToken in project cobertura:

```
public static class GTToken extends Token {
    int realKind = JavaParser15Constants.GT;
}
```

Byte code of class GTToken:

```
 0: aload_0
 1: invokespecial #10 // Method
          //  net/sourceforge/cobertura/javancss/parser/java15/Token."<init>":()V
 4: aload_0
 5: bipush        126
 7: putfield      #12   // Field realKind:I
10: return
```

We strongly suspect that this produces some difference in the results for down-call and external reuse and we also know that we report fewer cases than the authors. However, we can not report exactly what percent we miss. For example, we can not prove that all 7 % difference between the down-call results originates from this problem.

Finally, the limitation about non-system methods, also discussed in chapter 9, is also a threat to validity. We may see fewer number of subtype and external reuse pairs due to this limitation than the original study, but just like the cases discussed above, we can not estimate how much impact this limitation has on our results.

Chapter 13

Future Work

The future work can be done in two ways. For the replication study, it will consist of efforts to make the replication closer the original study. These are discussed in the first section. We also have some ideas about future work for the original study, especially about the proposed inheritance model and metrics. We discuss those ideas in the second section.

13.1 Replication Study

The replication study can be enhanced by eliminating limitations and threads to validity. Mainly, the problems about the interpretation of definitions and the differences between the analysed source code and byte code are the major issues to solve.

For the correct interpretation of each definition, a comparative study which can be conducted together with the authors of the original study is needed. A few pilot projects for investigation could be chosen and then the results for each inheritance category could be compared with each other. The possible differences would raise the necessary questions, and the answers would eliminate the interpretation differences, if any. Questions raised about the differences between the byte code and the source code of a given type (for example, the method invocations added to the byte code) may also be answered.

For an exact replication, the content of the source code and byte code should be the same. In our replication, this is for many projects, far from true. To solve this issue, we should first find ways of retrieving the content of the byte code properly. When we know the list of types in the byte code, for example, we may attempt to choose those types from the source code and set-up a source code package which is the same as (or at least similar to) the byte code contents. This can probably be achieved by an additional Rascal program.

The limitation about the external method analysis can also be solved. As the best option, Rascal language can be extended to enable analysis of non-system methods. Otherwise, heuristics we use can be enhanced to cover more cases.

We believe that the results of the original and replication studies can be very close to each other if these suggestions can be realized.

13.2 Original Study

The original study concentrates on the usage of the inheritance relationship. The programmers first define these relationships, probably with various purposes in mind. The WhaPDJI study examines what happens after this, in other words, given the defined inheritance relationships, they look at the usage of inheritance in the code.

As the name of the original study also suggests ("What Programmers Do With Inheritance in Java"), the intention is to look from the perspective of the programmer, who is actively using the defined inheritance relationships. Keeping this in mind, we have a few suggestions for a possible future direction for the original study:

13.2.1 Byte Code or Source Code Analysis

Because we want to understand what programmers are doing, analysis of the source code is perhaps a better choice than analysis of the byte code. Java programmers are busy with source code. The constructs added by the compiler (like method invocations) may be, however rare, misleading for the analysis.

13.2.2 Implicit or Explicit Relationships

In the proposed inheritance model, only the explicit pairs are taken in to account. To be able to analyse the indirect subtype and external reuse cases, the authors count all the pairs between the descendant and ascendant in the inheritance hierarchy as subtype and external reuse. This is an important choice which directly affects the study results. If we decide to choose another direction, and count both implicit and explicit inheritance relationships which have direct or indirect usages, we might have ended up in different results. The following simple example illustrates the consequence:

```
public class P {
    void p() {
        // ....
    }
}

public class C extends P { }

public class GC extends C { }

public class GGC extends GC { }

public class N {
    void n() {
        GGC aGGC = new GGC();
        aGGC.p();           // external reuse, between GGC and P
    }
}
```

In the current model, we have three explicit pairs: <GGC,GC> , <GC,C> and <C,P> . All those pairs take the external reuse attribute, because the external reuse attribute is propagated up in the inheritance hierarchy. According to the metrics, this simple system has 3 external reuse for 3 inheritance pairs, thus, 100 % usage.

However, if we also decide to count implicit relations, than there will be no need to propagate the attribute up in the hierarchy. We will than have 6 inheritance pairs (<GGC,GC> , <GGC,GC> , <GGC,C> , <GGC,P> , <GC,C> , <GC,P> and <C,P>) and one usage (between GGC and P). One usage, because the programmer intends to reuse method p() of type P from an object of type GGC. In this case, we will have about 17 % usage.

13.2.3 Down-call: Potential or Actual

Definition of down-call may also reflect the actual usage more than it is now. In the WhaPDJI study, the pairs of types which have a potential for down-call usage are counted as down-call pairs. We may argue that, if the programmer intends to actually issue a down-call, she or he would attempt to issue the method call of ascending type on an object of descendant type explicitly. In the following example:

```
public class P {
    void p() {
        q();
    }
    void q() {
    }
}

public class Q extends P {
    void q() {
    }
}

public class D {
    void d() {
        Q aQ = new Q();
        aQ.p();       // when p() is executed,
    }                 //  Q#q() is called instead of P#q()
}
```

the programmer attempts to make a down-call with the aQ.p() call. We believe that a down-call definition which includes such a method issue attempt will reflect the down-call usage by the programmer better than the current definition.

Moreover, this way of thinking (the programmer should attempt to use the method) is also used in the original study in reuse analysis. The authors do count the pairs which have a potential for external reuse only if a method or a field of the ascendant is actually accessed via an object of descendant type. The proposed down-call definition will also be more consistent with the rest of the definitions in the WhaPDJI study.

Since we would then introduce one more constraint, we may expect fewer down-calls than the original study.

13.2.4 One Reuse

It may be meaningful to discuss if the reuse analysis can be reduced to one reuse, instead of carrying out separate analyses for external and internal reuse. The research question about replacing inheritance with composition is about the reuse in general and not about internal or external reuse in particular. This may simplify the future analyses of reuse.

Chapter 14

Conclusion

In this study, we replicated an empirical software engineering study made by E. Tempero, H. Y. Yang and J. Noble about the various uses of inheritance in Java [TYN13]. We aimed for keeping our study set-up to the original study as close as possible, with one major difference. We analysed the source code whereas the byte code was analysed in the original study.

Our conclusions for the research questions are similar to the original study, but they are not the same. The first question is about how often down-call (late-bound self-reference) occurs in Java projects. Original study reports about one third of the inheritance relations involving such a case, while we found about one fourth. Their second research question is about the frequency of subtype usage. We found that about 60 % of all inheritance cases involve subtype relationship. The authors also report a higher percentage (66 %) about subtype usage. Third research question investigates the opportunity to replace inheritance with composition. All relationships which do not have subtype usage, but have reuse can be replaced by composition. Authors found a significant percentage of reuse (median 22 % for external and 2 % for internal reuse). We also report a similar percentage, but the division between internal and external uses is very different in our case (median of 3 % for external reuse and 20 % for internal reuse). For the last research question, which is about the other uses of inheritance in Java, the authors found out that these occur in many systems, but their use is not generally significant. Although our percentages are not the same with the original study for various other uses of inheritance, our results also agree with this conclusion.

When we investigate the possible reasons for the differences, we see that especially the differences in our study set up play a role here. For the subtype and external reuse, it is highly probable that we report fewer cases because of this. We also conclude that the analysis of byte code can result in false positives in some particular occasions. The down-call and external reuse results of the original study can become lower if these false positives are taken out.

Our replication study can be improved mainly by eliminating our limitations. Furthermore, collaboration with the authors about possible false positives can bring the results of the two studies even closer by.

As future work for the original study, one can discuss the proposed inheritance model and the metrics and consider some alternatives for detecting and counting various inheritance usages. Especially, how down-call usage is detected and the role of indirect usage in subtype and external reuse, according to us, present some opportunities for further discussion.

Bibliography

[Ayt14a] Cigdem Aytekin. Final Results Excel Sheet for replication of WhaPDJI study, 2014. URL: https://github.com/caytekin/Inheritance-Logs/blob/master/Final%20results/Final_Results_Thesis.xlsx.

[Ayt14b] Cigdem Aytekin. Inheritance Logs for Replication WhaPDJI Study, 2014. URL: https://github.com/caytekin/Inheritance-Logs.

[Ayt14c] Cigdem Aytekin. Rascal Code replication WhaPDJI Study, 2014. URL: https://github.com/caytekin/thesis-aytekin/tree/master/MasterThesis/src/inheritance.

[BRW^{+}08] A. Brooks, M. Roper, M. Wood, J. Daly, and J. Miller. Replication's role in software engineering. In Forrest Shull, Janice Singer, and DagI.K. Sjberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 365–379. Springer London, 2008. URL: http://dx.doi.org/10.1007/978-1-84800-044-5_14, doi:10.1007/978-1-84800-044-5_14.

[Car98] Michelle Cartwright. An empirical view of inheritance. *Information and Software Technology*, 40(14):795 – 799, 1998. URL: http://www.sciencedirect.com/science/article/pii/S0950584998001050, doi:http://dx.doi.org/10.1016/S0950-5849(98)00105-0.

[Car10] Jeffrey C Carver. Towards reporting guidelines for experimental replications: A proposal. In *International Workshop on Replication in Empirical Software Engineering Research, Cape Town, South Africa*, 2010.

[CK94] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

[DBM^{+}96] John Daly, Andrew Brooks, James Miller, Marc Roper, and Murray Wood. Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering*, 1(2):109–132, 1996. URL: http://dx.doi.org/10.1007/BF00368701, doi:10.1007/BF00368701.

[HS95] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1 edition, 1995.

[Lan13] Angelika Langer. Java Generics FAQs - Under The Hood Of The Compiler, 2013. URL: http://www.angelikalanger.com/GenericsFAQ/FAQSections/TechnicalDetails.html#FAQ100.

[LLPV11] Ralf Lammel, R. Linke, E. Pek, and A Varanovich. A framework profile of .NET. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 141–150, Oct 2011. doi:10.1109/WCRE.2011.25.

[MH90] Dennis Mancl and William Havanas. A study of the impact of C++ on software maintenance. In *Software Maintenance, 1990, Proceedings., Conference on*, pages 63–69. IEEE, 1990.

[NCS08] E. Nasseri, S. Counsell, and M. Shepperd. An empirical study of evolution of inheritance in Java OSS. In *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*, pages 269–278, March 2008. doi:10.1109/ASWEC.2008.4483215.

[SB08] Kathy Sierra and Bert Bates. *SCJP: Sun Certified Programmer for Java 6 Study Guide*. McGraw-Hill, 2008.

[SHH+05] D.IK. Sjoeberg, J.E. Hannay, O. Hansen, V.B. Kampenes, A Karahasanovic, N.-K. Liborg, and AC. Rekdal. A survey of controlled experiments in software engineering. *Software Engineering, IEEE Transactions on*, 31(9):733–753, Sept 2005. doi:10.1109/TSE.2005.97.

[TAD+08] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas Corpus Homepage. Curated Collection of Software Systems, 2008. URL: http://www.qualitascorpus.com/.

[TAD+10] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, December 2010. doi:10.1109/APSEC.2010.46.

[Tai96] Antero Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479, September 1996. URL: http://doi.acm.org/10.1145/243439.243441, doi:10.1145/243439.243441.

[Tem14] Ewan Tempero. personal e-mail communication, 2014.

[TMVB13a] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. Qualitas.class Corpus. (A Compiled Version of Qualitas Corpus), 2013. URL: http://java.labsoft.dcc.ufmg.br/qualitas.class/index.html.

[TMVB13b] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *Software Engineering Notes*, 38(5):1–4, 2013. doi:10.1145/2507288.2507314.

[TNM08] Ewan Tempero, James Noble, and Hayden Melton. How do Java programs use inheritance? An empirical study of inheritance in Java software. In Jan Vitek, editor, *ECOOP 2008 Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 667–691. Springer Berlin Heidelberg, 2008. URL: http://dx.doi.org/10.1007/978-3-540-70592-5_28, doi:10.1007/978-3-540-70592-5_28.

[TYN08] Ewan D. Tempero, Hong Yul Yang, and James Noble. Inheritance Use Data. Inheritance Use Data, 2008. URL: https://www.cs.auckland.ac.nz/~ewan/qualitas/studies/inheritance/.

[TYN13] Ewan D. Tempero, Hong Yul Yang, and James Noble. What programmers do with inheritance in Java. In Giuseppe Castagna, editor, *ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 577–601. Springer, 2013. doi:10.1007/978-3-642-39038-8_24.

[VRCG+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

Appendix A

Analysed Projects

The following table lists the projects analysed in the replication study. Three projects from WhaPDJI study could not be analysed because of errors: drjava-stable-20100913-r5387, jre-1.6.0 and roller-5.0.1.

ant-1.8.1 antlr-3.4 aoi-2.8.1 argouml-0.34 aspectj-1.6.9 axion-1.0-M2 c_jdbc-2.0.2 castor-1.3.3 cayenne-3.0.1 checkstyle-5.6 cobertura-1.9.4.1 colt-1.2.0 columba-1.0 derby-10.9.1.0 displaytag-1.2 drawswf-1.2.9 emma-2.0.5312 exoportal-v1.0.2 findbugs-1.3.9 fitjava-1.1 fitlibraryforfitnesse-20110301 freecol-0.10.3 freecs-1.3.20100406 galleon-2.3.0 ganttproject-2.1.1 heritrix-1.14.4 hibernate-4.2.0 hsqldb-2.2.0 htmlunit-2.8 informa-0.7.0-alpha2 ireport-3.7.5 itext-5.0.3 jFin_DateMath-R1.0.1 james-2.2.0 jasml-0.10 javacc-5.0 jchempaint-3.0.1 jedit-4.3.2 jext-5.0 jfreechart-1.0.13 jgraph-5.13.0.0 jgraphpad-5.10.0.2 jgrapht-0.8.1 jgroups-2.10.0 jhotdraw-7.5.1 jmeter-2.5.1 jmoney-0.4.4 joggplayer-1.1.4s jparse-0.96 jpf-1.5.1 jrat-1.0-beta1 jrefactory-2.9.19 jruby-1.7.3 jsXe-04_beta jspwiki-2.8.4 jtopen-7.8 jung-2.0.1 junit-4.1 log4j-2.0-beta lucene-4.2.0 marauroa-3.8.1 maven-3.0.5 megamek-0.35.18 mvnforum-1.2.2-ga myfaces_core-2.1.10 nakedobjects-4.0.0 nekohtml-1.9.14 openjms-0.7.7-beta-1 oscache-2.3 picocontainer-2.10.2 pmd-4.2.5 poi-3.6 pooka-3.0-080505 proguard-4.9 quickserver-1.4.7 quilt-0.6-a-5 rssowl-2.0.5 sablecc-3.2 springframework-3.0.5 squirrel_sql-3.2.1 struts-2.2.1 sunflow-0.07.2 tapestry-5.1.0.5 tomcat-7.0.2 trove-2.1.0 velocity-1.6.4 webmail-0.7.10 weka-3.6.9 xalan-2.7.1 xerces-2.10.0

Table A.1: List of analysed projects in the replication study

Appendix B

Version Differences

In the next table, the analysed versions which are different in the original and replication studies are listed:

Moreover, three projects from the Qualitas.class Corpus did not compile properly and we downloaded the source code from the open source websites like sourceforge: ant-1.8.1, squirell-3.2.1, velocity-1.6.4

Original Study	Replication Study
antlr-3.2	antlr-3.4
argouml-0.30.2	argouml-0.34
castor-1.3.1	castor-1.3.3
checkstyle-5.1	checkstyle-5.6
derby-10.6.1.0	derby-10.9.1.0
fitlibraryforfitnesse-20100806	fitlibraryforfitnesse-20110301
freecol-0.9.4	freecol-0.10.3
ganttproject-2.0.9	ganttproject-2.1.1
hibernate-3.6.0	hibernate-4.2.0
jmeter-2.4	jmeter-2.5.1
jpf-1.0.2	jpf-1.5.1
jrat-0.6	jrat-1.0-beta1
jruby-1.5.2	jruby-1.7.3
jtopen-7.1	jtopen-7.8
junit-4.8.2	junit-4.1
log4j-1.2.16	log4j-2.0-beta
lucene-2.4.1	lucene-4.2.0
maven-3.0	maven-3.0.5
myfaces_core-2.0.2	myfaces_core-2.1.10
oscache-2.4.1	oscache-2.3
proguard-4.5.1	proguard-4.9
sablecc-3.1	sablecc-3.2
springframework-1.2.7	springframework-3.0.5
squirrel-sql 3.1.2	squirrel-sql 3.2.1
weka-3.7.2	weka-3.6.9

Table B.1: Versions of the projects which are different between the original study and replication study.

Appendix C

Rascal Code

C.1 InternalReuse.rsc

```
module inheritance::InternalReuse

import IO;
import Map;
import Set;
import Relation;
import List;
import ListRelation;

import lang::java::m3::Core;
import lang::java::jdt::m3::Core;

import inheritance::InheritanceDataTypes;
import inheritance::InheritanceModules;



lrel [inheritanceKey, inheritanceSubtype, internalReuseDetail] getClassLevelInternalReuse(loc oneClass, set [loc] declaredFields,
     set [loc] declaredMethods,
                                              set [loc] ancestorFieldsMethods, map [loc, set [loc]] invocationMap, map [loc, set
                                                  [loc]] fieldAccessFromClassMap,
                                              map [loc, set [loc]] invertedContainmentMap) {
  lrel [inheritanceKey, inheritanceSubtype, internalReuseDetail] retRel = [];
  set [loc] invokedMethodsClassLevel  = oneClass in invocationMap ? invocationMap[oneClass] : {};
  set [loc] accessedFieldsClassLevel  = oneClass in fieldAccessFromClassMap ? fieldAccessFromClassMap[oneClass] : {};
  set [loc] internalReuseLocs     =   {aMethod | aMethod <- invokedMethodsClassLevel, isMethod(aMethod), aMethod notin
      declaredMethods,  aMethod in ancestorFieldsMethods }
                          +
                          {aField | aField <- accessedFieldsClassLevel, isField(aField), aField notin declaredFields, aField in
                              ancestorFieldsMethods };
  for (reusedLoc <- internalReuseLocs) {
    loc parentClass =  getDefiningClassOfALoc(reusedLoc, invertedContainmentMap);
    inheritanceKey iKey = <oneClass, parentClass>;
    retRel += < iKey, INTERNAL_REUSE_CLASS_LEVEL, <reusedLoc, oneClass>>;
  };
  return retRel;
}



public rel [inheritanceKey, inheritanceType] getInternalReuseCases(M3 projectM3) {
  // Get all the child classes in CC relation, they are the only ones which can initiate internal reuse.
  // The super() calls in the constructors are not counted as internal reuse, also see assumptions document.
  //
  // In classes, not only the methods, but also the initializers are taken into account
  rel [inheritanceKey, inheritanceType] resultRel = {};
  lrel [inheritanceKey, inheritanceSubtype, internalReuseDetail] internalReuseLog = [];
  rel [loc, loc] allInheritanceRels = getNonFrameworkInheritanceRels(getInheritanceRelations(projectM3), projectM3);
  set [loc] intReuseClasses = { child | <child, parent> <- allInheritanceRels, isClass(child), isClass(parent)};
  map [loc, set [loc]] containmentMap     = toMap({<owner, declared> |<owner, declared> <- projectM3@containment, isClass(owner),
       isField(declared) || isMethod(declared)});
  map [loc, set [loc]] containmentMapWithInit = toMap({<owner, declared> |<owner, declared> <- projectM3@containment, isClass(
      owner), isField(declared) || isMethod(declared) || declared.scheme == "java+initializer"});
  map [loc, set [loc]] invertedContainmentMap = toMap(invert({<owner, declared> |<owner, declared> <- projectM3@containment,
      isClass(owner), isField(declared) || isMethod(declared) || declared.scheme == "java+initializer"}));
  map [loc, set [loc]] invocationMap      = toMap({ <caller, invoked> | <caller, invoked> <- projectM3@methodInvocation});
  map [loc, set [loc]] fieldAccessFromMethodMap     = toMap({<accessor, accessed> | <accessor, accessed> <- projectM3@fieldAccess
      , isMethod(accessor) || accessor.scheme == "java+initializer"});
  map [loc, set [loc]] fieldAccessFromClassMap    = toMap({<accessor, accessed> | <accessor, accessed> <- projectM3@fieldAccess,
      isClass(accessor)});

  for (oneClass <- intReuseClasses) {
    set [loc] ancestorClasses = { parent | <child, parent> <- allInheritanceRels, child == oneClass, isClass(parent)};
    set [loc] ancestorFieldsMethods = {};
    for (anAncestorClass <- ancestorClasses) {
      ancestorFieldsMethods += anAncestorClass in containmentMap ? containmentMap[anAncestorClass] : {};
    }
    set [loc] declaredFieldsMethods = oneClass in containmentMapWithInit ? containmentMapWithInit[oneClass] : {} ;
    set [loc] declaredMethods = { declMeth | declMeth <- declaredFieldsMethods, isMethod(declMeth) || declMeth.scheme == "java+
        initializer" };
    set [loc] declaredFields = declaredFieldsMethods - declaredMethods;

    internalReuseLog += getClassLevelInternalReuse(oneClass, declaredFields, declaredMethods,   ancestorFieldsMethods,
        invocationMap,
                                                    fieldAccessFromClassMap, invertedContainmentMap );
    for (oneMethod <- declaredMethods) {
      set [loc] allInvokedMethods = oneMethod in invocationMap ? invocationMap[oneMethod] : {};

      set [loc] internalReuseMethodInvocation = { invoked | invoked <- allInvokedMethods, invoked notin declaredFieldsMethods,
                                        invoked in ancestorFieldsMethods,
                                        invoked.scheme != "java+constructor" };
      set [loc] allAccessedFields = oneMethod in fieldAccessFromMethodMap ? fieldAccessFromMethodMap[oneMethod] : {} ;

      set [loc] internalReuseFieldAccess = { accessed | accessed <- allAccessedFields,  accessed notin declaredFieldsMethods,
                                                  accessed in ancestorFieldsMethods };
      set [loc] internalReuseLoc = internalReuseMethodInvocation + internalReuseFieldAccess;
      for (reusedLoc <- internalReuseLoc) {
        loc parentClass =  getDefiningClassOfALoc(reusedLoc, invertedContainmentMap);
        inheritanceKey iKey = <oneClass, parentClass>;
        internalReuseLog += < iKey, INTERNAL_REUSE_METHOD_LEVEL, <reusedLoc, oneMethod>>;
      };
    };
  };

  resultRel = {<iKey, INTERNAL_REUSE> | <iKey, _, <_,_>> <- internalReuseLog};
  iprintToFile(getFilename(projectM3.id, internalReuseLogFile), internalReuseLog);
  println("Size␣of␣internal␣reuse␣log␣is:␣<size(internalReuseLog)>");
  return resultRel;
}
```

C.2 ExternalReuse.rsc

```
module inheritance::ExternalReuse

import IO;
import Map;
import Set;
import Relation;
import List;
import ListRelation;
import Node;
import ValueIO;

import util::ValueUI;

import lang::java::m3::Core;
import lang::java::m3::AST;
import lang::java::jdt::m3::Core;
import lang::java::m3::TypeSymbol;

import inheritance::InheritanceDataTypes;
import inheritance::InheritanceModules;





public loc getImmParentForAccess(loc classOrInterfaceOfReceiver, loc accessedFieldOrMethod,
                                  map [loc, set[loc]]   invClassAndInterfaceContainment,
                                  map [loc, set[loc]]   declarationsMap,
                                  rel [loc, loc]      allInheritanceRelations,
                                  map [loc, set[loc]]   extendsMap,
                                  map [loc, set[loc]]   implementsMap,
                        M3 projectM3) {
  loc immediateParentOfReceiver = DEFAULT_LOC;
  bool direct = false;

  if (isLocDefinedInProject(classOrInterfaceOfReceiver, declarationsMap) && ! (isLocDefinedInGivenType(accessedFieldOrMethod,
      classOrInterfaceOfReceiver, invClassAndInterfaceContainment))) {   // external reuse
    if (isLocDefinedInProject(accessedFieldOrMethod,  declarationsMap)) {
      loc locDefiningClassOrInterface = getDefiningClassOrInterfaceOfALoc(accessedFieldOrMethod, invClassAndInterfaceContainment,
           projectM3);
      immediateParentOfReceiver = getImmediateParentGivenAnAsc(classOrInterfaceOfReceiver, locDefiningClassOrInterface,
          extendsMap, implementsMap, allInheritanceRelations);
    }
    else {
      immediateParentOfReceiver = getImmediateParent(classOrInterfaceOfReceiver, extendsMap, implementsMap, declarationsMap);
    }
  }
  return immediateParentOfReceiver ;
}



public lrel [inheritanceKey, inheritanceSubtype, loc, loc] getExternalReuseGeneral (loc classOfAccess, TypeSymbol recTypeSymbol,
    loc accessedLoc, loc srcRef,
                                            map [loc, set[loc]]   invClassAndInterfaceContainment,
                                            map [loc, set[loc]]   declarationsMap,
                                            rel [loc, loc]      allInheritanceRelations,
                                            map[loc, set[loc]]    extendsMap,
                                            map[loc, set[loc]]    implementsMap,
                                            M3 projectM3) {
  lrel [inheritanceKey, inheritanceSubtype, loc, loc] retList = [];
  loc classOrInterfaceOfReceiver  = getClassOrInterfaceFromTypeSymbol(recTypeSymbol);
  immediateParentOfReceiver = getImmParentForAccess(classOrInterfaceOfReceiver, accessedLoc,  invClassAndInterfaceContainment,
      declarationsMap,
                                        allInheritanceRelations, extendsMap, implementsMap, projectM3);
  if (immediateParentOfReceiver != DEFAULT_LOC) { // external reuse
```

```
    if ((isLocDefinedInProject(accessedLoc, declarationsMap)) && isLocDefinedInGivenType(accessedLoc, immediateParentOfReceiver, invClassAndInterfaceContainment)) {
      if (classOfAccess != classOrInterfaceOfReceiver) {
        retList += <<classOrInterfaceOfReceiver, immediateParentOfReceiver>, EXTERNAL_REUSE_DIRECT, srcRef, accessedLoc>;
      }
    }
    else {
      if (isLocDefinedInProject(accessedLoc, declarationsMap) ) {
        loc definingTypeOfLoc = getDefiningClassOrInterfaceOfALoc(accessedLoc, invClassAndInterfaceContainment, projectM3);
        lrel [loc, loc] inhChain = getInheritanceChainGivenAsc( classOrInterfaceOfReceiver, definingTypeOfLoc,  extendsMap, implementsMap,  declarationsMap, allInheritanceRelations);
        for (aPair <- inhChain) {
          retList += <aPair, EXTERNAL_REUSE_INDIRECT, srcRef, accessedLoc>;
        }
      }
      else {  // LIMITATION! If the accessedLoc is defined outside of the project, we just insert the immediate parent and nothing else in between.
        retList += <<classOrInterfaceOfReceiver, immediateParentOfReceiver>, EXTERNAL_REUSE_INDIRECT, srcRef, accessedLoc>;
      }
    }
  }

  return retList;
}






public lrel [inheritanceKey, inheritanceSubtype, loc, loc] getExternalReuseViaMethodCall(Expression mCall, loc  classOfMethodCall,
                                            map [loc, set[loc]]   invClassAndInterfaceContainment,
                                            map [loc, set[loc]]   declarationsMap,
                                            rel [loc, loc]      allInheritanceRelations,
                                            map[loc, set[loc]]    extendsMap,
                                            map[loc, set[loc]]    implementsMap,
                                            M3 projectM3) {
  lrel [inheritanceKey, inheritanceSubtype, loc, loc] retList = [];
  loc methodDefiningClassOrInterface = DEFAULT_LOC;
  visit (mCall) {
    case m2:\methodCall(_, receiver:_, _, _): {
      loc invokedMethod = m2@decl;
      if (invokedMethod == |unresolved:///|) {
        appendToFile(getFilename(projectM3.id, errorLog), "In␣getExternalReuseViaMethodCall,␣methodcall␣decl␣unresolved␣for␣method␣call:␣<invokedMethod>,␣at␣<m2@src>.␣Receiver␣is:␣<receiver>\n\n");
        println("Methodcall␣decl␣unresolved␣for␣method␣call:␣<invokedMethod>,␣at␣<m2@src>.␣Receiver␣is:␣<receiver>");
      }
      else {
        retList = getExternalReuseGeneral (classOfMethodCall, receiver@typ, invokedMethod, m2@src,
            invClassAndInterfaceContainment,  declarationsMap, allInheritanceRelations, extendsMap, implementsMap, projectM3);
      } // else
      } // case methodCall()
    }  // visit
  return retList;
}


public lrel [inheritanceKey, inheritanceSubtype, loc, loc] getExternalReuseViaFieldAccess(Expression qName, loc
    classOfFieldAccess,
                                              map [loc, set[loc]]   invClassAndInterfaceContainment,
                                              map [loc, set[loc]]   declarationsMap,
                                              rel [loc, loc]      allInheritanceRelations,
                                              map[loc, set[loc]]    extendsMap,
                                              map[loc, set[loc]]    implementsMap,
                                              M3 projectM3) {
  lrel [inheritanceKey, inheritanceSubtype, loc, loc] retList = [];
  loc accessedField = DEFAULT_LOC;
  TypeSymbol receiverTypeSymbol = DEFAULT_TYPE_SYMBOL;
  loc srcRef = DEFAULT_LOC;
  bool isThisReference = false;
  visit (qName) {
    case qName:\qualifiedName(qualifier, expression) : {
      accessedField = expression@decl;
      if (isField(accessedField) ) { receiverTypeSymbol = qualifier@typ; }
      srcRef = expression@src;
      fieldReceiver = qualifier@decl;
    }
    case fAccessStmt:\fieldAccess(isSuper, fAccessExpr:_, str name:_) : {
      accessedField = fAccessStmt@decl;
      receiverTypeSymbol = fAccessExpr@typ;
      srcRef = fAccessStmt@src;
      isThisReference = (fAccessExpr := this());
    }
  }  // visit
  if (isField(accessedField) && !(isThisReference) ) {
    retList = getExternalReuseGeneral (classOfFieldAccess, receiverTypeSymbol, accessedField, srcRef,
        invClassAndInterfaceContainment,  declarationsMap, allInheritanceRelations, extendsMap, implementsMap, projectM3);
  } // if
  return retList;
}

private map [metricsType, num]  calculateExternalReuseIndirectPercentages(  rel [inheritanceKey, inheritanceType]
    explicitFoundInhrels,
                                      rel [inheritanceKey, inheritanceType]  addedImplicitRels, M3 projectM3) {
  map [metricsType, num] addedResultsMap = ();
  addedResultsMap += (perAddedCCExtReuse : getPercentageAdded(explicitFoundInhrels, addedImplicitRels, EXTERNAL_REUSE, "java+
      class", "java+class"));
  addedResultsMap += (perAddedCIExtReuse : getPercentageAdded(explicitFoundInhrels, addedImplicitRels, EXTERNAL_REUSE, "java+
      class", "java+interface"));
  addedResultsMap += (perAddedIIExtReuse : getPercentageAdded(explicitFoundInhrels, addedImplicitRels, EXTERNAL_REUSE, "java+
      interface", "java+interface"));
  for (aMetric <- [perAddedCCExtReuse, perAddedCIExtReuse, perAddedIIExtReuse]) {
    println("<getNameOfInheritanceMetric(aMetric)>␣:␣<addedResultsMap[aMetric]>");
    appendToFile(getFilename(projectM3.id, addedPercentagesFile), "<addedResultsMap[aMetric]>␣\t");
  }
  return addedResultsMap;
}




public rel [inheritanceKey, inheritanceType] getExternalReuseCases(M3 projectM3) {
  rel [inheritanceKey, inheritanceType] resultRel = {};
  lrel [inheritanceKey, inheritanceSubtype, loc, loc] allExternalReuseCases = [];
  lrel [inheritanceKey, inheritanceType, loc, loc] allCandExtReuseCases = [];
  set [loc]         allClassesInProject     = {decl | <decl, prjct> <- projectM3@declarations, isClass(decl) };
  map [loc, set [loc]]  containmentMapForMethods  = toMap({<owner, declared> | <owner,declared> <- projectM3@containment, isClass
      (owner), isMethod(declared)});
  map [loc, set [loc]]  invertedClassContainment  = toMap(invert({<owner, declared> | <owner,declared> <- projectM3@containment,
      isClass(owner)}));
  map [loc, set [loc]]  invClassAndInterfaceContainment   = getInvertedClassAndInterfaceContainment(projectM3);
  map [loc, set [loc]]  invClassInterfaceMethodContainment  = getInvertedClassInterfaceMethodContainment(projectM3);
  map [loc, set [loc]]  declarationsMap       = toMap({<aLoc, aProject> | <aLoc, aProject> <- projectM3@declarations});
  map [loc, set [loc]]  extendsMap    = toMap({<_child, _parent> | <_child, _parent> <- projectM3@extends});
  map [loc, set [loc]]    implementsMap   = toMap({<_child, _parent> | <_child, _parent> <- projectM3@implements});
  rel [loc, loc]      allInheritanceRelations   = getInheritanceRelations(projectM3);
  map [loc, set[loc]]   invertedUnitContainment   = getInvertedUnitContainment(projectM3);
  for (oneClass <- allClassesInProject) {
    println("External␣reuse␣cases␣for␣class:␣<oneClass>␣are␣going␣to␣be␣collected...");
    list [Declaration] ASTsOfOneClass = getASTsOfAClass(oneClass, invClassInterfaceMethodContainment, invertedUnitContainment,
        declarationsMap, projectM3);
    for (oneAST <- ASTsOfOneClass) {
      visit(oneAST) {
            case qName:\qualifiedName(_, _) : {
              allExternalReuseCases += getExternalReuseViaFieldAccess(qName, oneClass, invClassAndInterfaceContainment,
                  declarationsMap, allInheritanceRelations, extendsMap, implementsMap, projectM3);
            }
            case fAccess:\fieldAccess(_,_,_) : {
              allExternalReuseCases += getExternalReuseViaFieldAccess(fAccess, oneClass, invClassAndInterfaceContainment,
                  declarationsMap, allInheritanceRelations, extendsMap, implementsMap, projectM3);
            }
        case m2:\methodCall(_, receiver:_, _, _): {
          allExternalReuseCases += getExternalReuseViaMethodCall(m2, oneClass, invClassAndInterfaceContainment, declarationsMap,
                allInheritanceRelations, extendsMap, implementsMap, projectM3);
              } // case methodCall()
            } // visit()
      } // for each method in the class
    } // for each class in the project
    rel [inheritanceKey, inheritanceType] directExReusePairs = {};
    rel [inheritanceKey, inheritanceType] indirectExReusePairs = {};
    for ( int i <- [0..size(allExternalReuseCases)]) {
      tuple [ inheritanceKey iKey, inheritanceSubtype iType, loc srcLoc, loc accessedLoc] aCase = allExternalReuseCases[i];
      if (aCase.iType == EXTERNAL_REUSE_DIRECT) { directExReusePairs += <aCase.iKey, EXTERNAL_REUSE> ; }
      if (aCase.iType == EXTERNAL_REUSE_INDIRECT) { indirectExReusePairs += <aCase.iKey, EXTERNAL_REUSE> ; }
      resultRel += <aCase.iKey, EXTERNAL_REUSE>;
    }
    calculateExternalReuseIndirectPercentages(directExReusePairs, indirectExReusePairs, projectM3);
    iprintToFile(getFilename(projectM3.id, externalReuseLogFile), allExternalReuseCases);
    return resultRel;
}
```

C.3 SubtypeInheritance.rsc

```
module inheritance::SubtypeInheritance

import IO;
import Map;
import Set;
import Relation;
import List;
import ListRelation;
import Node;
import ValueIO;

import lang::java::m3::Core;
import lang::java::m3::AST;
import lang::java::jdt::m3::Core;
import lang::java::m3::TypeSymbol;

import inheritance::InheritanceDataTypes;
import inheritance::InheritanceModules;



public list [TypeSymbol] getPassedSymbolList(Expression methExpr, M3 projectM3) {
    list [Expression] args    = [];
    list [TypeSymbol] retList   = [];
    visit (methExpr)  {
      case \methodCall(_,_,myArgs:_) : {
        args = myArgs;
    }
    case \methodCall(_,_,_,myArgs:_) : {
      args = myArgs;
    }
    case newObject1:\newObject(Type \type, list[Expression] expArgs) : {
      args = expArgs;
    }
    case newObject2:\newObject(Type \type, list[Expression] expArgs, Declaration class) : {
      args = expArgs;
    }
    case newObject3:\newObject(Expression expr, Type \type, list[Expression] expArgs) : {
      args = expArgs;
    }
    case newObject4:\newObject(Expression expr, Type \type, list[Expression] expArgs, Declaration class) : {
      args = expArgs;
    }
    case consCall1:\constructorCall(bool isSuper, list[Expression] expArgs) : {
      args = expArgs;
    }
    case consCall2:\constructorCall(bool isSuper, Expression expr, list[Expression] expArgs) : {
      args = expArgs;
    }
  }
  TypeSymbol argTypeSymbol = DEFAULT_TYPE_SYMBOL;
  for ( int i <- [0..(size(args))]) {
    argTypeSymbol = getTypeSymbolFromAnnotation(args[i], projectM3);
    if (argTypeSymbol == DEFAULT_TYPE_SYMBOL) {
      println("No␣such␣symbol␣annotation␣for␣:␣Method␣expression:␣<methExpr>,␣args:␣<args>");
      retList = [];
      break;
    }
    retList += argTypeSymbol;
  }
  return retList;
}


private lrel [inheritanceKey, inheritanceSubtype , loc]  getSubtypeResultViaAssignment(Expression lhs, Expression rhs, loc
    sourceRef, M3 projectM3) {
  lrel [inheritanceKey, inheritanceSubtype , loc ] retList = [];
  TypeSymbol lhsTypeSymbol = getTypeSymbolFromAnnotation(lhs, projectM3);
  TypeSymbol rhsTypeSymbol = getTypeSymbolFromAnnotation(rhs, projectM3);
  if ( (lhsTypeSymbol == DEFAULT_TYPE_SYMBOL) || (rhsTypeSymbol == DEFAULT_TYPE_SYMBOL)) {
    println("In␣getSubtypeResultViaAssignment,␣NoSuchAnnotation␣exception␣thrown␣for:␣lhs:␣:␣<lhs>,␣or␣rhs:␣<rhs>␣at␣sourceref␣:␣
        <sourceRef>");
  }
  else {
    tuple [bool isSubtypeRel, inheritanceKey iKey] result = getSubtypeRelation(rhsTypeSymbol, lhsTypeSymbol);
    if (result.isSubtypeRel) {
      retList += <result.iKey, SUBTYPE_ASSIGNMENT_STMT, sourceRef>;
    } // if
  }
  return retList;
}



public lrel [inheritanceKey, inheritanceSubtype , loc ] getSubtypeViaAssignment(Expression asmtStmt, M3 projectM3) {
  lrel [inheritanceKey, inheritanceSubtype , loc ] retList = [];
  bool isConditional = false;
  visit (asmtStmt) {
    case aStmt:\assignment(lhs, operator, rhs) : {
      visit (aStmt) {
        case conditionalS:\conditional(logicalExpr, thenBranch, elseBranch) : {   // ternary operator, ,ex: p4 = (i == 3) ? new C() : new G();
          isConditional = true;
          retList += getSubtypeResultViaAssignment(lhs, thenBranch, conditionalS@src, projectM3);
          retList += getSubtypeResultViaAssignment(lhs, elseBranch, conditionalS@src, projectM3);
        }
      }
      if (!isConditional) {
        retList += getSubtypeResultViaAssignment(lhs, rhs, aStmt@src, projectM3);
      }
    } // case
  } // visit
  return retList;
}


private lrel [inheritanceKey, inheritanceSubtype , loc ] getSubtypeResultViaVariable(TypeSymbol lhsTypeSymbol, Expression rhs, list [Expression] fragments, M3 projectM3) {
  lrel [inheritanceKey, inheritanceSubtype , loc ] retList = [];
  TypeSymbol rhsTypeSymbol = getTypeSymbolFromAnnotation(rhs, projectM3);
  if (rhsTypeSymbol != DEFAULT_TYPE_SYMBOL) {
    tuple [bool isSubtypeRel, inheritanceKey iKey] result = getSubtypeRelation(rhsTypeSymbol, lhsTypeSymbol);
    if (result.isSubtypeRel) {
      for (anExpression <- fragments) {
        retList += <result.iKey, SUBTYPE_ASSIGNMENT_VAR_DECL, anExpression@decl>;
      }
    }
  }
  return retList;
}


public lrel [inheritanceKey, inheritanceSubtype , loc ]  getSubtypeViaVariables(Type typeOfVar, list[Expression] fragments, M3
    projectM3) {
  lrel [inheritanceKey, inheritanceSubtype , loc ] retList = [];
  bool isConditional = false;
  TypeSymbol lhsTypeSymbol = getTypeSymbolFromRascalType(typeOfVar);
  visit (fragments[size(fragments) - 1]) {
    case nullVar: \variable(_, _ , null()) : {
      // a null initialization should not be counted as subtype
    ;}
    case myVar: \variable(_,_,stmt) : {
      visit (stmt) {
        case conditionalS:\conditional(logicalExpr, thenBranch, elseBranch) : {
          isConditional = true;
          retList += getSubtypeResultViaVariable(lhsTypeSymbol , thenBranch, fragments, projectM3);
          retList += getSubtypeResultViaVariable(lhsTypeSymbol , elseBranch, fragments, projectM3);
        }
      }
      if (!isConditional) {
        retList += getSubtypeResultViaVariable(lhsTypeSymbol , stmt, fragments, projectM3);
      }
    } // case myVar
    }  // visit fragments
  return retList;
}





private bool isUpCasting(loc aChild, loc aParent, map [loc, set [loc]] inheritanceRelsMap ) {
  bool retBool = false;
  if ( (aChild == OBJECT_CLASS) && (aParent != OBJECT_CLASS) ) {
    retBool = true;
  }
  else {
    set [loc] allParentsOfAChild = (aChild in inheritanceRelsMap) ? inheritanceRelsMap[aChild] : {};
    if (aParent notin allParentsOfAChild) {
      // we suspect upcasting
      set [loc] reverseParentSet = (aParent in inheritanceRelsMap) ? inheritanceRelsMap[aParent] : {};
      if (aChild in reverseParentSet) {
        retBool = true;
      }
    }
  ;}
  if (retBool) {  println("upcasting␣between␣<aChild>␣and␣<aParent>....."); }
  return retBool;
}
```

```
private tuple [bool , loc ] isSidewaysCast(loc aChild, loc aParent, map [loc, set [loc]] inheritanceRelsMap, map [loc, set [loc]] invertedInheritanceRelsMap,   loc castLoc) {
  bool isSideCast = false;
  loc implClass = DEFAULT_LOC;
  set [loc] allParentsOfAChild = (aChild in inheritanceRelsMap) ? inheritanceRelsMap[aChild] : {};
  set [loc] allParentsOfAParent = (aParent in inheritanceRelsMap) ? inheritanceRelsMap[aParent] : {};
  if ( isInterface(aParent) && isInterface(aChild) && (aParent notin allParentsOfAChild) && (aChild notin allParentsOfAParent)) {
    // we suspect sideways casting, find the implementing class
    set [loc] allImplClassesOfParent = (aParent in invertedInheritanceRelsMap) ? invertedInheritanceRelsMap[aParent] : {};
    set [loc] allImplClassesOfChild  = (aChild in invertedInheritanceRelsMap) ? invertedInheritanceRelsMap[aChild] : {};
    set [loc] implClasses = allImplClassesOfParent & allImplClassesOfChild;
    if (!isEmpty(implClasses)) {
      isSideCast = true;
      implClass = getOneFrom(implClasses);
    }
    else {
      // possibly non sytem (framework) types.
    ;}
  }
  return <isSideCast, implClass>;
}



public lrel [inheritanceKey, inheritanceSubtype , loc ] getSubtypeViaCast(Expression castStmt, map [loc, set [loc]] inheritanceRelsMap, map [loc, set [loc]] invertedInheritanceRelsMap,
                                                        M3 projectM3 ) {
// The cast can be from child to parent, but also from parent to child
// Also, sideways cast is possible.
  lrel [inheritanceKey, inheritanceSubtype , loc ] retList = [];
  visit (castStmt) {
    case \cast(castType, castExpr) : {
      TypeSymbol castExprTypeSymbol = getTypeSymbolFromAnnotation(castExpr, projectM3);
      if (castExprTypeSymbol != DEFAULT_TYPE_SYMBOL) {
        tuple [bool isSubtypeRel, inheritanceKey iKey] result = getSubtypeRelation(getTypeSymbolFromRascalType(castType), castExprTypeSymbol);
        if (result.isSubtypeRel) {
          bool upcasting = isUpCasting(result.iKey.child, result.iKey.parent, inheritanceRelsMap);
          if (upcasting) {
            // reverse the order if there is upcasting.
            retList += < <result.iKey.parent, result.iKey.child> , SUBTYPE_VIA_UPCASTING, castStmt@src>;
          }
          else {
            if (isInterface(result.iKey.child) && isInterface (result.iKey.parent) ) {
              tuple [bool isSidewaysCast, loc implChild] sidewaysResult = isSidewaysCast(result.iKey.child, result.iKey.parent, inheritanceRelsMap, invertedInheritanceRelsMap, castStmt@src);
              if (sidewaysResult.isSidewaysCast) {
                retList += <<sidewaysResult.implChild, result.iKey.child>, SUBTYPE_VIA_SIDEWAYS_CASTING, castStmt@src>;
                retList += <<sidewaysResult.implChild, result.iKey.parent>, SUBTYPE_VIA_SIDEWAYS_CASTING, castStmt@src>;
              }
              else {
                retList += <result.iKey, SUBTYPE_VIA_CAST, castStmt@src>;
              }
            }
            else {
              retList += <result.iKey, SUBTYPE_VIA_CAST, castStmt@src>;
            }
          }
        }
      }
    } // case
  } // visit
  return retList;
}


private TypeSymbol getTypeSymbolFromTypeParameterList(TypeSymbol collTypeSymbol, loc forLoopRef, list [TypeSymbol] typeParameters, M3 projectM3) {
  TypeSymbol retSymbol = DEFAULT_TYPE_SYMBOL;
  if (size(typeParameters) == 0) { retSymbol = OBJECT_TYPE_SYMBOL; }
  else {
    if (size(typeParameters) == 1)  {
      retSymbol = typeParameters[0];
    }
    else {
      appendToFile(getFilename(projectM3.id, errorLog), "More␣than␣one␣type␣parameter␣in␣class/interface␣collection␣def:␣<collTypeSymbol>␣at␣<forLoopRef>.␣Type␣parameters:␣<typeParameters>\n\n");
      println("More␣than␣one␣type␣parameter␣in␣class/interface␣collection␣def:␣<collTypeSymbol>␣at␣<forLoopRef>.␣Type␣parameters:␣<typeParameters>");
    }
  }
  return retSymbol;
}



private lrel [inheritanceKey, inheritanceSubtype , loc ] getSubtypeResultViaForLoop(TypeSymbol paramTypeSymbol, Expression collExpression, loc forLoopRef, M3 projectM3) {
  lrel [inheritanceKey, inheritanceSubtype , loc ] retList = [];
  TypeSymbol compTypeSymbol = DEFAULT_TYPE_SYMBOL;
  TypeSymbol collTypeSymbol = getTypeSymbolFromAnnotation(collExpression,  projectM3);
  if (collTypeSymbol != DEFAULT_TYPE_SYMBOL) {
    switch (collTypeSymbol) {
      case anArray:\array(TypeSymbol component, int dimension) : {
        compTypeSymbol = component;
      }
      case anInterfaceColl:\class(loc decl, list[TypeSymbol] typeParameters) : {
        compTypeSymbol = getTypeSymbolFromTypeParameterList(collTypeSymbol, forLoopRef, typeParameters, projectM3);
      }
      case aClassColl:\interface(loc decl, list[TypeSymbol] typeParameters) : {
        compTypeSymbol = getTypeSymbolFromTypeParameterList(collTypeSymbol, forLoopRef, typeParameters, projectM3);
      }
    }
    if (compTypeSymbol != DEFAULT_TYPE_SYMBOL) {
      tuple [bool isSubtypeRel, inheritanceKey iKey] result = getSubtypeRelation(compTypeSymbol, paramTypeSymbol);
      if (result.isSubtypeRel) { retList += <result.iKey, SUBTYPE_VIA_FOR_LOOP, forLoopRef>; }
    }
  }
  return retList;
}


public lrel [inheritanceKey, inheritanceSubtype , loc ] getSubtypeViaReturnStmt(Statement returnStmt, loc methodLoc,map [loc, set [TypeSymbol]] typesMap, M3 projectM3 ) {
  lrel [inheritanceKey, inheritanceSubtype , loc ] retList = [];
  TypeSymbol retTypeSymbol = DEFAULT_TYPE_SYMBOL;
  visit (returnStmt) {
    case \return(retExpr) : {
      retTypeSymbol = getTypeSymbolFromAnnotation(retExpr, projectM3);
      if (retTypeSymbol != DEFAULT_TYPE_SYMBOL) {
        tuple [bool isSubtypeRel, inheritanceKey iKey] result = getSubtypeRelation(retTypeSymbol,
                                            getDeclaredReturnTypeSymbolOfMethod(methodLoc, typesMap));
        if (result.isSubtypeRel) {
          retList += <result.iKey, SUBTYPE_VIA_RETURN, retExpr@src>;
        }
      }
    }
  }
  return retList;
}


private TypeSymbol getTypeSymbolOfVararg(TypeSymbol varargSymbol) {
  TypeSymbol retSymbol = DEFAULT_TYPE_SYMBOL;
  if (array(TypeSymbol component, int dimension) := varargSymbol ) {
    retSymbol = component;
  }
  else {
    throw ("In␣getTypesymbolOfVararg,␣the␣vararg␣is␣not␣of␣type␣array␣:␣<varargSymbol>");
  }
  return retSymbol;
}
```

```
private bool isVararg(TypeSymbol passedSymbol, TypeSymbol declaredSymbol, rel [loc, loc] allInheritanceRelations) {
  bool retBool = false;
  if ( array(TypeSymbol component, int dimension) := declaredSymbol ) {
    loc declaredLoc = getClassOrInterfaceFromTypeSymbol(declaredSymbol);
    loc passedLoc   = getClassOrInterfaceFromTypeSymbol(passedSymbol);
    if (inheritanceRelationExists(passedLoc, declaredLoc, allInheritanceRelations)) {
      retBool = true;
    }
  }
  return retBool;
}



private list [TypeSymbol] updateDeclaredSymbolListForVararg(list [TypeSymbol] passedSymbolList, list [TypeSymbol] declaredSymbolList, rel [loc, loc] allInheritanceRelations, M3 projectM3 ) {
  list [TypeSymbol] retList = [];
  if ( size(passedSymbolList) == size(declaredSymbolList) ) {
    retList = declaredSymbolList;
    if ( size(passedSymbolList) > 0 ) {
      if ( last(passedSymbolList) != last(declaredSymbolList) ) {
        if (isVararg(last(passedSymbolList), last(declaredSymbolList), allInheritanceRelations)) {
          retList[size(retList)-1] = getTypeSymbolOfVararg(last(declaredSymbolList));
        }
      }
    }
  }
  else {
    if (size(passedSymbolList) < size(declaredSymbolList) ) {
      retList = prefix(declaredSymbolList);
    }
    else {
      if (size(passedSymbolList) > size(declaredSymbolList)) {
        retList = prefix(declaredSymbolList);
        int numOfElementsToAdd = size(passedSymbolList) - size(declaredSymbolList);
        if (isVararg(last(passedSymbolList), last(declaredSymbolList), allInheritanceRelations)) {
          TypeSymbol typeSymbolToAdd = getTypeSymbolOfVararg(declaredSymbolList[size(declaredSymbolList) - 1] );
          for (int i <- [0..numOfElementsToAdd+1]) { retList +=  typeSymbolToAdd ; }
        }
        else {
          appendToFile(getFilename(projectM3.id, errorLog), "In␣updateDeclaredSymbolListForVararg,␣the␣vararg␣could␣not␣be␣analyzed.␣Passed␣symbol␣list:␣<passedSymbolList>,␣declared␣symbol␣list:␣<declaredSymbolList>␣\n\n");
          println("In␣updateDeclaredSymbolListForVararg,␣the␣vararg␣could␣not␣be␣analyzed.␣Passed␣symbol␣list:␣<passedSymbolList>,␣declared␣symbol␣list:␣<declaredSymbolList>␣");
          retList = [];
        }
      }
    }
  }
  return retList;
}



public lrel [inheritanceKey, inheritanceSubtype , loc ] getSubtypeViaParameterPassing(Expression methOrConstExpr, map [loc, set[
    loc]] declarationsMap,
                            map [loc, set[TypeSymbol]]  typesMap,
                            map [loc, set[loc]]     invertedClassAndInterfaceContainment,
                            map [loc, set [str]]    invertedNamesMap,
                            rel [loc, loc]        allInheritanceRelations,
                            M3 projectM3) {
  lrel [inheritanceKey, inheritanceSubtype , loc ]  retList = [];
  bool analyzeMethod = false;
  list [TypeSymbol] finalDeclaredSymbolList = [];
  list [TypeSymbol] passedSymbolList    = getPassedSymbolList(methOrConstExpr, projectM3);
  if (methOrConstExpr@decl in typesMap) {
    analyzeMethod = true;
    list [TypeSymbol] declaredSymbolList  = getDeclaredParameterTypes(methOrConstExpr, typesMap,
        invertedClassAndInterfaceContainment, projectM3);
    finalDeclaredSymbolList         = updateDeclaredSymbolListForVararg(passedSymbolList, declaredSymbolList,
        allInheritanceRelations,  projectM3);
  }
  else {  // method is not defined in the source, we try to get the method parameters wuth a heuristic.
    str methodStr = (methOrConstExpr@decl).file;
    finalDeclaredSymbolList = getArgTypeSymbols(methodStr, invertedNamesMap);
  }
  if (!isEmpty(finalDeclaredSymbolList)) {analyzeMethod = true;}
  else { analyzeMethod = false; }
  if (analyzeMethod) {
    if (size(finalDeclaredSymbolList) < size(passedSymbolList)) { // the number of declared method arguments is less than the
        number of passed parameters
      retList = [];
    }
    else {
      for (int i <- [0..size(passedSymbolList)]) {
        tuple [bool isSubtypeRel, inheritanceKey iKey] result = getSubtypeRelation(passedSymbolList[i], finalDeclaredSymbolList[i
            ]);
        if (result.isSubtypeRel) {
          retList +=  <result.iKey, SUBTYPE_VIA_PARAMETER, methOrConstExpr@src>;
        }
      }
    }
  }
  return retList;
}

```

```
public rel [inheritanceKey, inheritanceType] getSubtypeCases(M3 projectM3) {
  rel [inheritanceKey, inheritanceType]       resultRel   = {};
  lrel [inheritanceKey, inheritanceSubtype , loc ]      subtypeLog  = [];
  lrel [inheritanceKey, inheritanceSubtype , loc ]      allSubtypeCases = [];
  set [loc]         allClassesAndInterfacesInProject  = getAllClassesAndInterfacesInProject(projectM3);
  set [loc]         allClassesInProject         = getAllClassesInProject(projectM3);
  map [loc, set [loc]]  declarationsMap           = toMap({<aLoc, aProject> | <aLoc, aProject> <- projectM3@declarations});
  map [loc, set [loc]]  methodsOfClasses            = toMap({<owner, declared> | <owner,declared> <- projectM3@containment,
      isClass(owner), isMethod(declared)});
  map [loc, set[TypeSymbol]] typesMap             = toMap({<from, to> | <from, to> <- projectM3@types});
  map [loc, set[loc]]   invertedUnitContainment       = getInvertedUnitContainment(projectM3);
  map [loc, set[loc]]   invertedClassAndInterfaceContainment = getInvertedClassAndInterfaceContainment(projectM3);
  map [loc, set[loc]]   invertedClassInterfaceMethodContainment = getInvertedClassInterfaceMethodContainment (projectM3);
  rel [loc, loc]      projectInhRels             = getInheritanceRelations(projectM3);
  map [loc, set [loc]]  inheritanceRelsMap          = toMap(projectInhRels);
  map [loc, set [loc]]  invertedInheritanceRelsMap      = toMap(invert(projectInhRels));
  map [loc, set [loc]]  extendsMap              = toMap({<_child, _parent> | <_child, _parent> <- projectM3@extends});
  map [loc, set [loc]]    implementsMap             = toMap({<_child, _parent> | <_child, _parent> <- projectM3@implements});
  map [loc, set [str]]  invertedNamesMap           = getInvertedNamesMap(projectM3@names);
  rel [loc, loc]      allInheritanceRelations       = getInheritanceRelations(projectM3);
  for (oneClass <- allClassesInProject ) {
    list [Declaration] ASTsOfOneClass = getASTsOfAClass(oneClass, invertedClassInterfaceMethodContainment ,
        invertedUnitContainment, declarationsMap, projectM3);
    for (oneAST <- ASTsOfOneClass) {
      visit(oneAST) {
        case aStmt:\assignment(lhs, operator, rhs) : {
          allSubtypeCases += getSubtypeViaAssignment(aStmt, projectM3);
        }
        case _variables:\variables(typeOfVar, fragments) : {
          allSubtypeCases += getSubtypeViaVariables(typeOfVar, fragments, projectM3);
        }
        case _fields:\field(typeOfField, fragments) : {
          allSubtypeCases += getSubtypeViaVariables(typeOfField, fragments, projectM3);
        }
        case castStmt:\cast(castType, castExpr) : {
          allSubtypeCases += getSubtypeViaCast(castStmt, inheritanceRelsMap, invertedInheritanceRelsMap, projectM3);
        } // case \cast
        case enhForStmt:\foreach(Declaration parameter, Expression collection, Statement body) : {
          allSubtypeCases += getSubtypeResultViaForLoop(parameter@typ, collection, enhForStmt@src, projectM3);
        }
        case  methExpr1:\methodCall(_,_, _) : {
          allSubtypeCases +=  getSubtypeViaParameterPassing(methExpr1, declarationsMap, typesMap,
              invertedClassAndInterfaceContainment, invertedNamesMap, allInheritanceRelations, projectM3);
        }
        case methExpr2:\methodCall(_, _, _, _): {
          allSubtypeCases +=  getSubtypeViaParameterPassing(methExpr2, declarationsMap, typesMap,
              invertedClassAndInterfaceContainment, invertedNamesMap, allInheritanceRelations, projectM3);
        }
        case newObject1:\newObject(Type \type, list[Expression] expArgs) : {
          allSubtypeCases +=  getSubtypeViaParameterPassing(newObject1, declarationsMap, typesMap,
              invertedClassAndInterfaceContainment, invertedNamesMap, allInheritanceRelations, projectM3);
        }
        case newObject2:\newObject(Type \type, list[Expression] expArgs, Declaration class) : {
          allSubtypeCases +=  getSubtypeViaParameterPassing(newObject2, declarationsMap, typesMap,
              invertedClassAndInterfaceContainment, invertedNamesMap, allInheritanceRelations, projectM3);
        }
        case newObject3:\newObject(Expression expr, Type \type, list[Expression] expArgs) : {
          allSubtypeCases +=  getSubtypeViaParameterPassing(newObject3, declarationsMap, typesMap,
              invertedClassAndInterfaceContainment, invertedNamesMap, allInheritanceRelations, projectM3);
        }
        case newObject4:\newObject(Expression expr, Type \type, list[Expression] expArgs, Declaration class) : {
          allSubtypeCases +=  getSubtypeViaParameterPassing(newObject4, declarationsMap, typesMap,
              invertedClassAndInterfaceContainment, invertedNamesMap, allInheritanceRelations, projectM3);
        }
        case consCall1:\constructorCall(bool isSuper, list[Expression] arguments) : {
          Expression consCallExpr1 = createMethodCallFromConsCall(consCall1);
          allSubtypeCases +=  getSubtypeViaParameterPassing(consCallExpr1, declarationsMap, typesMap,
              invertedClassAndInterfaceContainment, invertedNamesMap, allInheritanceRelations, projectM3);
        }
        case consCall2:\constructorCall(bool isSuper, Expression expr, list[Expression] arguments) : {
          Expression consCallExpr2 = createMethodCallFromConsCall(consCall2);
          allSubtypeCases +=  getSubtypeViaParameterPassing(consCallExpr2, declarationsMap, typesMap,
              invertedClassAndInterfaceContainment, invertedNamesMap, allInheritanceRelations, projectM3);
        }

          } // visit()
    }

    set [loc] methodsInClass = oneClass in methodsOfClasses ? methodsOfClasses[oneClass] : {};
    for (oneMethod <- methodsInClass) {
      methodAST = getMethodASTEclipse(oneMethod, model = projectM3);
      visit(methodAST) {
        case returnStmt:\return(retExpr) : {
          allSubtypeCases += getSubtypeViaReturnStmt(returnStmt, oneMethod, typesMap, projectM3 );
        }  // case \return(_)
          } // visit()
    } // for each method in the class
  } // for each class in the project
  for ( int i <- [0..size(allSubtypeCases)]) {
    tuple [ inheritanceKey iKey, inheritanceSubtype iType, loc detLoc] aCase = allSubtypeCases[i];
    resultRel += <<aCase.iKey.child, aCase.iKey.parent>, SUBTYPE> ;
    subtypeLog += aCase;
  }
  iprintToFile(getFilename(projectM3.id,subtypeLogFile),subtypeLog );
  return resultRel;
}
```

C.4 ThisChangingType.rsc

```
module inheritance::ThisChangingType

import IO;
import Map;
import Set;
import Relation;
import List;
import ListRelation;
import Node;
import ValueIO;
import DateTime;

import lang::java::m3::Core;
import lang::java::m3::AST;
import lang::java::jdt::m3::Core;
import lang::java::jdt::m3::AST;
import lang::java::m3::TypeSymbol;

import inheritance::InheritanceDataTypes;
import inheritance::InheritanceModules;



private set [loc] surfaceMatch(Expression aStatement, list [Expression] args) {
  set [loc] retSet = {};
  for (anArg <- args) {
    if (this() := anArg) {retSet += aStatement@src; }
  }
  return retSet;
}


private set [loc] getThisReferencesInAST(Declaration anAST) {
  set [loc] retSet = {};
  visit (anAST) {
    case methExpr2:\methodCall(_,_,_,_args) : {
      retSet += surfaceMatch(methExpr2, _args);
    }
    case aStmt:\assignment(_lhs, _operator , _rhs) : {
      if (_rhs := this()) {
        retSet += aStmt@src;
      }
    }
    case variables:\variables(_, _fragments) : {
      visit (_fragments[0]) {
        case vrb:\variable(_,_,_rhs) : {
          if (_rhs := this()) {
            retSet += _fragments[0]@src;
          }
        }
      }
    }
    case newObject1:\newObject(Type \type, list[Expression] expArgs) : {
      retSet += surfaceMatch(newObject1, expArgs);
    }
    case newObject2:\newObject(Type \type, list[Expression] expArgs, Declaration class) : {
      retSet += surfaceMatch(newObject2, expArgs);
    }
    case newObject3:\newObject(Expression expr, Type \type, list[Expression] expArgs) : {
      retSet += surfaceMatch(newObject3, expArgs);
    }
    case newObject4:\newObject(Expression expr, Type \type, list[Expression] expArgs, Declaration class) : {
      retSet += surfaceMatch(newObject4, expArgs);
    }
    case methExpr1:\methodCall(_,_,_args) : {
      retSet += surfaceMatch(methExpr1, _args);
    }
  }
  return retSet;
}


private set [loc]  getThisReferencesInMethod(loc aMethodOfAscClass, M3 projectM3) {
  Declaration methodAST = getMethodASTEclipse(aMethodOfAscClass, model = projectM3);
  set [loc] retSet = getThisReferencesInAST(methodAST);
  return retSet;
}


private set [loc] getThisReferencesInClass(loc ascClass, map [loc, set [loc]] invertedClassInterfaceMethodContainment, map [loc, set [loc]] invertedUnitContainment,
                                                          map [loc, set [loc]] declarationsMap, M3 projectM3) {
  list [Declaration] ASTsOfOneClass = getASTsOfAClass(ascClass, invertedClassInterfaceMethodContainment, invertedUnitContainment, declarationsMap, projectM3);
  set [loc] retSet = {};
  for (anAST <- ASTsOfOneClass ) {
    retSet += getThisReferencesInAST(anAST);
  }
  return retSet;
}



private rel [loc, loc, loc] getThisChangingTypeCandidates(M3 projectM3) {
  // I only look for class - class relationships
  // ascClass, descClass, candMethod
  rel [loc, loc, loc] retRel = {};
  rel [inheritanceKey, thisChangingTypeCandDetail] candidateLog = {};
  rel [loc _desc, loc _asc]   allInheritanceRels = getNonFrameworkInheritanceRels(getInheritanceRelations(projectM3), projectM3);
  map [loc, set[loc]]     ascDescPair = toMap(invert({<_desc, _asc> | <_desc, _asc> <- allInheritanceRels, isClass(_desc), isClass(_asc)}));
  map [loc, set [loc]]    methodsInClasses = toMap({<_aClass, _aMethod> | <_aClass, _aMethod> <- projectM3@containment, isClass(_aClass), isMethod(_aMethod)});
  map [loc, set[loc]]     invertedContainment = getInvertedClassAndInterfaceContainment(projectM3);
  map [loc, set[loc]]     methodClassContainment = toMap({<_aClass, _aMethod> | <_aClass, _aMethod> <- projectM3@containment, isClass(_aClass), isMethod(_aMethod)});
  map [loc, set [loc]]    invertedClassAndInterfaceContainment = getInvertedClassAndInterfaceContainment(projectM3);
  map [loc, set[loc]]     invertedClassInterfaceMethodContainment = getInvertedClassInterfaceMethodContainment(projectM3);
  map [loc, set[loc]]     invertedUnitContainment = getInvertedUnitContainment(projectM3);
  map [loc, set[loc]]     declarationsMap =  toMap({<aLoc, aProject> | <aLoc, aProject> <- projectM3@declarations});
  map[loc, set[loc]]      extendsMap = toMap({<_child, _parent> | <_child, _parent> <- projectM3@extends});
  for (ascClass <- ascDescPair) {
    set [loc] allDescClassesOfAscClass = ascDescPair[ascClass];
    for (descClass <- allDescClassesOfAscClass ) {
      set [loc] allMethodsOfAscClass =  ascClass in methodClassContainment ? methodClassContainment[ascClass] : {};
      set [loc] allCandidateRefsInMethods = {};
      for (aMethodOfAscClass <- allMethodsOfAscClass) {
        set[loc] candidateMethodReferences = getThisReferencesInMethod(aMethodOfAscClass, projectM3);
        allCandidateRefsInMethods += candidateMethodReferences;
        if (!isMethodOverriddenByAnyDescClass(aMethodOfAscClass, ascClass, descClass, invertedContainment, extendsMap, projectM3) ) {
          for (aCandRef <- candidateMethodReferences ) {
            candidateLog += <<descClass, ascClass>, <aMethodOfAscClass, aCandRef>>;
            retRel += <descClass, ascClass, aMethodOfAscClass>;
          }
        } // if
      } // for aMethodOfAscClass
      // look at the initializers of ascending class.
      set [loc] candidateClassReferences = getThisReferencesInClass(ascClass, invertedClassInterfaceMethodContainment, invertedUnitContainment, declarationsMap, projectM3);
      set [loc] candidateInitializerReferences = candidateClassReferences - allCandidateRefsInMethods;
      for (anInitRef <- candidateInitializerReferences) {
        candidateLog += <<descClass, ascClass>, <ascClass, anInitRef>>;
        retRel += <descClass, ascClass, ascClass>;
      }
    } // for  descClass
  } // for ascClass
  iprintToFile(getFilename(projectM3.id,thisChangingTypeCandFile), candidateLog);
  return retRel;
}


private loc getAscClassFromDescAndMethod(loc invokedMethod, loc descClass, rel [loc, loc, loc] candidates) {
  loc retLoc = DEFAULT_LOC;
  set [loc] ascClasses = {_ascClass| <_descClass, _ascClass, _calledMethod> <- candidates, _descClass == descClass, _calledMethod
       == invokedMethod };
  if (size(ascClasses) != 1) {
    throw ("In␣getAscClassFromDescAndMethod,␣the␣size␣of␣ascClasses␣is␣not␣1,␣but␣<size(ascClasses)>.␣ascClasses:␣<ascClasses>");
  }
  retLoc = getOneFrom(ascClasses);
  return retLoc;
}


private rel [inheritanceKey, thisChangingTypeOccurrence] getOccurrenceItems(Expression newObjExpr, Type oType, rel [loc, loc]
    initializerThisRefClasses,
                                        set [loc] initializerChildren) {
  rel [inheritanceKey, thisChangingTypeOccurrence] occurrenceItems = {};
  TypeSymbol newTypeSymbol = getTypeSymbolFromRascalType(oType);
  loc newClass = getClassFromTypeSymbol(newTypeSymbol);
  if ((newClass != DEFAULT_LOC) && (newClass in initializerChildren)) {
    set [inheritanceKey] keySet = {<_child, _parent> | <_child, _parent> <- initializerThisRefClasses, _child == newClass};
    for (inheritanceKey iKey <- keySet ) {
      occurrenceItems += <iKey, <newObjExpr@src, newClass>>;
    }
  }
  return occurrenceItems;
}


rel [inheritanceKey, thisChangingTypeOccurrence] getAnOccurrenceFromMethodCall( rel [loc, loc, loc] candidates,       loc
    invokedMethod,
                                        map [loc, set[loc]] methodAndDescClasses,   loc receivingClass,
                                        loc srcRef) {
  rel [inheritanceKey, thisChangingTypeOccurrence] retOccurrence = {};
  if (invokedMethod in methodAndDescClasses) {
    set [loc] candDescClasses = methodAndDescClasses[invokedMethod];
    if (receivingClass in candDescClasses) {
      loc ascClass = getAscClassFromDescAndMethod(invokedMethod, receivingClass, candidates);
      retOccurrence += <<receivingClass, ascClass>, <srcRef, invokedMethod>>;
    }
  }
  return retOccurrence;
}



rel [inheritanceKey, thisChangingTypeOccurrence] getThisChangingTypeOccurencesFromTheChild(rel [loc, loc, loc] candidates, M3 projectM3) {
  rel [inheritanceKey, thisChangingTypeOccurrence] occurrenceLog = {};
  map [loc, set[loc]]   methodAndDescClasses          = toMap({<calledMethod, descClass> | <descClass, ascClass, calledMethod> <- candidates, isMethod(calledMethod)});
  set [loc]         allClassesInProject           = getAllClassesInProject(projectM3);
  map [loc, set[loc]]   invertedClassInterfaceMethodContainment = getInvertedClassInterfaceMethodContainment (projectM3);
  map [loc, set[loc]]   invertedUnitContainment         = getInvertedUnitContainment(projectM3);
  map [loc, set [loc]]  declarationsMap             = toMap({<aLoc, aProject> | <aLoc, aProject> <- projectM3@declarations});
  for (oneClass <- allClassesInProject ) {
    list [Declaration] ASTsOfOneClass = getASTsOfAClass(oneClass, invertedClassInterfaceMethodContainment , invertedUnitContainment, declarationsMap, projectM3);
    for (oneAST <- ASTsOfOneClass) {
      visit(oneAST) {
        case  methExpr2:\methodCall(_,_, _) : {
          occurrenceLog += getAnOccurrenceFromMethodCall(candidates, methExpr2@decl, methodAndDescClasses, oneClass, methExpr2@src);
        }
        case consCall1:\constructorCall(_, _, _) : {
          occurrenceLog += getAnOccurrenceFromMethodCall(candidates, consCall1@decl, methodAndDescClasses, oneClass, consCall1@src);
        }
          case consCall2:\constructorCall(_, _) : {
          occurrenceLog += getAnOccurrenceFromMethodCall(candidates, consCall2@decl, methodAndDescClasses, oneClass, consCall2@src);
          }
          } // visit()
    } // for oneAST
  } // for oneClass
  return occurrenceLog;
}



private set [inheritanceKey] getThisChangingTypeOccurrences(rel [loc, loc, loc] candidates, M3 projectM3) {
  rel [inheritanceKey, thisChangingTypeOccurrence] occurrenceLog = {};
  set [inheritanceKey] retSet = {};
  map [loc, set[loc]] methodAndDescClasses = toMap({<calledMethod, descClass> | <descClass, ascClass, calledMethod> <- candidates, isMethod(calledMethod)});
  rel [loc, loc] initializerThisRefClasses = {<_descClass, _ascClass> | <_descClass, _ascClass, _classRef> <- candidates, isClass(_classRef)};
  set [loc] initializerThisRefChildren = domain(initializerThisRefClasses);
  loc aProject = projectM3.id;
  set [Declaration] projectASTs = createAstsFromEclipseProject(aProject, true);
  for (aProjectAST <- projectASTs) {
    visit (aProjectAST) {
      case methExpr1:\methodCall(_,receiver,_,_) : {
      // \methodCall(bool isSuper, Expression receiver, str name, list[Expression] arguments)
        TypeSymbol receiverTypeSymbol = getTypeSymbolFromAnnotation(receiver, projectM3);
        if (receiverTypeSymbol != DEFAULT_TYPE_SYMBOL) {
            loc classOfReceiver = getClassFromTypeSymbol(receiver@typ);
          occurrenceLog += getAnOccurrenceFromMethodCall(candidates, methExpr1@decl, methodAndDescClasses, classOfReceiver,
              methExpr1@src);
        }
        } // case \methodCall
      case methExpr2:\methodCall(_,_,_) : {
      //\methodCall(bool isSuper, str name, list[Expression] arguments)
        // This is handled separately in getThisChangingTypeOccurencesFromTheChild
      ;}
      case newObject1:\newObject(Type oType:\type, list[Expression] expArgs) : {
        occurrenceLog += getOccurrenceItems(newObject1, oType, initializerThisRefClasses, initializerThisRefChildren);
      }
      case newObject2:\newObject(Type oType:\type, list[Expression] expArgs, Declaration class) : {
        occurrenceLog += getOccurrenceItems(newObject2, oType, initializerThisRefClasses, initializerThisRefChildren);
      }
      case newObject3:\newObject(Expression expr, Type oType:\type, list[Expression] expArgs) : {
        occurrenceLog += getOccurrenceItems(newObject3, oType, initializerThisRefClasses, initializerThisRefChildren);
      }
      case newObject4:\newObject(Expression expr, Type oType:\type, list[Expression] expArgs, Declaration class) : {
        occurrenceLog += getOccurrenceItems(newObject4, oType, initializerThisRefClasses, initializerThisRefChildren);
      }
    }
  }
  occurrenceLog += getThisChangingTypeOccurencesFromTheChild(candidates, projectM3) ;
  retSet = { <_descClass, _ascClass> | <<_descClass, _ascClass>, <_srcRef, _methodOrClass>> <- occurrenceLog };
  iprintToFile(getFilename(projectM3.id,thisChangingTypeOccurFile), occurrenceLog);
  return retSet;
}


public rel [inheritanceKey, inheritanceType] getThisChangingTypeOccurrences(M3 projectM3) {
  rel [loc, loc, loc] thisChangingTypeCandidates = getThisChangingTypeCandidates(projectM3);
  set [inheritanceKey] thisChangingTypeOccurrences = getThisChangingTypeOccurrences(thisChangingTypeCandidates, projectM3);
  return {<iKey, SUBTYPE> | iKey <- thisChangingTypeOccurrences };
}
```

C.5 DowncallCases.rsc

```
module inheritance::DowncallCases

```

```
import IO;
import Map;
import Set;
import Relation;
import List;
import ListRelation;
import Node;
import ValueIO;

import lang::java::m3::Core;
import lang::java::m3::AST;
import lang::java::jdt::m3::Core;
import lang::java::m3::TypeSymbol;

import inheritance::InheritanceDataTypes;
import inheritance::InheritanceModules;


private tuple [bool, loc] isDowncall(loc invokedMethod, loc classOfReceiver,  loc refToDowncall, rel [loc, loc, loc, loc] downcallCandidates,
                                        rel [loc, loc] allInheritanceRels, map[loc, set[loc]] extendsMap) {
  bool retBool = false;
  loc theIssuerMethod= DEFAULT_LOC, descDowncalledMethod = DEFAULT_LOC;
  rel [loc, loc, loc, loc] downcallSet = {<_ascClass, _descClass, _ascIssuerMethod, _descDowncalledMethod> | <_ascClass, _descClass, _ascIssuerMethod, _descDowncalledMethod> <- downcallCandidates,
                                    invokedMethod == _ascIssuerMethod,
                                    classOfReceiver == _descClass ||
                                    classOfReceiver in getDescendantsOfAClass(_descClass, allInheritanceRels)
                                    };
  if (size(downcallSet) >= 1) {
    tuple [loc _ascClass, loc _descClass, loc _ascIssuerMethod, loc _descDowncalledMethod] downcallCase = getOneFrom(downcallSet);
    theIssuerMethod = downcallCase._ascIssuerMethod;
    if (size(downcallSet) == 1) {
      descDowncalledMethod = downcallCase._descDowncalledMethod;
    }
    else {
      list [loc] ascendantsInOrder = getAscendantsInOrder(classOfReceiver, extendsMap);
      for ( aNode <- ascendantsInOrder) {
        set [loc] anotherDowncallSet = {_descDowncalledMethod | <_ascClass, _descClass, _ascIssuerMethod, _descDowncalledMethod> <- downcallSet,
                                _ascIssuerMethod == invokedMethod,    // added on 19-5-2014
                                _descClass == aNode};
        if (size(anotherDowncallSet) == 1) {
          descDowncalledMethod = getOneFrom(anotherDowncallSet);
          break;
        }
        else {
          if (size(anotherDowncallSet) > 1) {
            // There are more than one descDowncalledmethod's. I put an arbitrarily slected one to the log
            descDowncalledMethod = getOneFrom(anotherDowncallSet);
            break;
          }
        }
      }
    }
    retBool = true;
  }
  return <retBool, descDowncalledMethod>;
}


map [loc, set[loc]] getInvertedOverridesMap (M3 projectM3) {
  map [loc, set[loc]] retMap = ();
  set [loc] allMethodsInProject     = {definedMethod | <definedMethod, project> <- projectM3@declarations, isMethod(definedMethod)};
  rel [loc, loc] allOverriddenMethods = {<descMeth, ascMeth> | <descMeth, ascMeth> <- projectM3@methodOverrides, ascMeth in allMethodsInProject};
  if (!isEmpty(allOverriddenMethods)) {
    retMap = toMap(invert(allOverriddenMethods));
  }
  return retMap;
}


loc getDescOverridingMethod(set [loc] overridingMethods, loc _descClass,map [loc, set[loc]] invertedContainment) {
  loc retMethod     = DEFAULT_LOC;
  int numOfMethods  = size(overridingMethods);
  list [loc] methodList   = toList(overridingMethods);
  int i = 0;
  bool found  = false;
  while ( ( i < numOfMethods) && (!found) ) {
    loc aMethod = methodList[i];
    loc definingClassOfAMethod = getDefiningClassOfALoc(aMethod, invertedContainment);
    if (definingClassOfAMethod == _descClass) {
      retMethod = aMethod;
      found = true;
    }
    i = i + 1;
  }
  return retMethod;
}


private rel [loc, loc, loc, loc] getDowncallCandidatesFromInitializers(map[loc, set[loc]] invertedClassAndInterfaceContainment ,
    M3 projectM3) {
```

```
  rel [loc ascendingClass, loc descendingClass, loc initializer, loc descDowncalledMethod] initializerCandidates = {};
  map [loc, set[loc]] invertedUnitContainment = getInvertedUnitContainment(projectM3);
  map [loc, set[loc]] invertedContainment   = getInvertedClassAndInterfaceContainment(projectM3);
  map [loc, set[loc]] invertedClassInterfaceMethodContainment = getInvertedClassInterfaceMethodContainment(projectM3);
  map [loc, set[loc]] declarationsMap     = toMap({<aLoc, aProject> | <aLoc, aProject> <- projectM3@declarations});
  set [loc]       allClassesInProject   = getAllClassesInProject(projectM3);
  rel [loc, loc]    inhClassPairs       = { <_descClass, _ascClass> | <_descClass, _ascClass> <- getInheritanceRelations(
      projectM3),  _descClass in allClassesInProject ,
                                                                     _ascClass in allClassesInProject };
  map [ loc, set[loc]] invertedOverridesMap =  getInvertedOverridesMap (projectM3);
  for ( <_descClass, _ascClass> <- inhClassPairs){
    list [Declaration] astsOfAscClass = getASTsOfAClass(_ascClass, invertedClassInterfaceMethodContainment,
        invertedUnitContainment, declarationsMap, projectM3);
    for (anAST <-astsOfAscClass) {
      visit (anAST) {
        case anInitializer:\initializer(Statement initializerBody) : {
          visit (anInitializer) {
            case m1:\methodCall(_,_,_) : {
              invokedMethod = m1@decl;
              set [loc] overridingMethods = invokedMethod in invertedOverridesMap ? invertedOverridesMap[invokedMethod] : {};
              loc overridingDescMethod = getDescOverridingMethod(overridingMethods, _descClass, invertedContainment);
              if (overridingDescMethod != DEFAULT_LOC) {
                initializerCandidates += <_ascClass, _descClass, anInitializer@decl, overridingDescMethod>;
              }  // if
            }  // case methodCall
            case m2:\methodCall(_,Expression receiver,_,_) : {
              if (receiver := this()) {
                invokedMethod = m2@decl;
                set [loc] overridingMethods = invokedMethod in invertedOverridesMap ? invertedOverridesMap[invokedMethod] : {};
                loc overridingDescMethod = getDescOverridingMethod(overridingMethods, _descClass, invertedContainment);
                if (overridingDescMethod != DEFAULT_LOC) {
                  initializerCandidates += <_ascClass, _descClass, anInitializer@decl, overridingDescMethod>;
                }  // if
              }
            ;} // case m2
          } // visit (initializer)
        } // case initializer
      } // visit AST
    } // for anAST
  } // for _descClass, _ascClass
  return initializerCandidates;
}


private rel [loc, loc, loc, loc] getDowncallCandidates(map[loc, set[loc]] invertedClassAndInterfaceContainment , M3 projectM3) {
  // Constructors are also included in the candidate analysis.
  rel [loc ascendingClass, loc descendingClass, loc ascIssureMethod, loc descDowncalledMethod] downcallCandidates = {};
  set [loc]         allMethodsInProject     = {definedMethod | <definedMethod, project> <- projectM3@declarations, isMethod(
    definedMethod)};
  rel [loc, loc]      allOverriddenMethods    = {<descMeth, ascMeth> | <descMeth, ascMeth> <- projectM3@methodOverrides, ascMeth in allMethodsInProject};
  map [loc, set [loc]]  containmentMapForMethods  = toMap({<owner, declared> | <owner,declared> <- projectM3@containment, isClass(owner), isMethod(declared)});
  for (<descMeth, ascMeth> <- allOverriddenMethods) {
    loc ascClass        = getDefiningClassOrInterfaceOfALoc(ascMeth, invertedClassAndInterfaceContainment, projectM3 );
    loc descClass         = getDefiningClassOrInterfaceOfALoc(descMeth, invertedClassAndInterfaceContainment, projectM3 );
    if ((ascClass != DEFAULT_LOC) && (descClass != DEFAULT_LOC)) {
      set [loc] methodsInAscClass = ascClass in  containmentMapForMethods ? containmentMapForMethods[ascClass] : {};
      for (issuerMethod <- methodsInAscClass) {
        methodAST = getMethodASTEclipse(issuerMethod, model = projectM3);
        visit(methodAST) {
          case mCall1:\methodCall(_,_,_) : {
            if ((mCall1@decl == ascMeth) && !isMethodOverriddenByDescClass(issuerMethod, descClass, invertedClassAndInterfaceContainment , projectM3)) {
              downcallCandidates += <ascClass, descClass, issuerMethod, descMeth >;
            }
           }
           case mCall2:\methodCall(_, Expression receiver, _, _) : {
            if (receiver := this() && (mCall2@decl == ascMeth) && !isMethodOverriddenByDescClass(issuerMethod, descClass, invertedClassAndInterfaceContainment , projectM3)) {
              downcallCandidates += <ascClass, descClass, issuerMethod, descMeth >;
            }
           }
        } // visit
      } // for
    } // if
  }
  downcallCandidates += getDowncallCandidatesFromInitializers(invertedClassAndInterfaceContainment, projectM3);
  println("Number␣of␣down␣call␣candidates␣for␣project:␣<size(downcallCandidates)>");
  return downcallCandidates;
}


public rel [inheritanceKey, inheritanceType] getDowncallOccurrences(M3 projectM3) {
  // Downcall is only possible between class class edges, by definition.
  map [loc, set[loc]]   invertedClassAndInterfaceContainment = getInvertedClassAndInterfaceContainment(projectM3);
  map [loc, set[loc]]   invertedClassInterfaceMethodContainment = getInvertedClassInterfaceMethodContainment(projectM3);
  map [loc, set [loc]]  containmentMapForMethods  = toMap({<owner, declared> | <owner,declared> <- projectM3@containment, isClass(owner), isMethod(declared)});
  map [loc, set [loc]]  extendsMap          = toMap({<_child, _parent> | <_child, _parent> <- projectM3@extends});
  rel [loc, loc]      allInheritanceRels      = getNonFrameworkInheritanceRels(getInheritanceRelations(projectM3), projectM3);
  map [loc, set[loc]]   invertedUnitContainment   = getInvertedUnitContainment(projectM3);
  map [loc, set [loc]]  declarationsMap       = toMap({<aLoc, aProject> | <aLoc, aProject> <- projectM3@declarations});
  rel [loc ascendingClass, loc descendingClass, loc ascIssuerMethod, loc descDowncalledMethod] downcallCandidates = getDowncallCandidates(invertedClassAndInterfaceContainment , projectM3);
  lrel [inheritanceKey, downcallDetail] downcallLog = [];
  rel [inheritanceKey, inheritanceType] resultRel = {};
  set [loc] allClassesInProject = getAllClassesInProject(projectM3);
  for (oneClass <- allClassesInProject ) {
    list [Declaration] ASTsOfOneClass = getASTsOfAClass(oneClass, invertedClassInterfaceMethodContainment,
        invertedUnitContainment, declarationsMap, projectM3);
    for (oneAST <- ASTsOfOneClass) {
      visit(oneAST) {
        case mCall1:\methodCall(_, receiver:_, _, _): {
          loc invokedMethod = mCall1@decl;
          TypeSymbol receiverTypeSymbol = getTypeSymbolFromAnnotation(receiver,  projectM3);
          if (receiverTypeSymbol != DEFAULT_TYPE_SYMBOL) {
            loc classOfReceiver = getClassFromTypeSymbol(receiverTypeSymbol);
            tuple [bool downcallBool, loc descDCallMeth] downcallResult = isDowncall(invokedMethod, classOfReceiver, mCall1@src,
                downcallCandidates, allInheritanceRels, extendsMap);
            if (downcallResult.downcallBool) {
              downcallLog += <<classOfReceiver, getDefiningClassOrInterfaceOfALoc(invokedMethod,
                  invertedClassAndInterfaceContainment, projectM3 )>, <mCall1@src, invokedMethod, downcallResult.descDCallMeth>>;
            }
          }
        }
        case mCall2:\methodCall(_,_,_) : {
          loc invokedMethod = mCall2@decl;
          loc classOfReceiver = oneClass;
          tuple [bool downcallBool, loc descDCallMeth] downcallResult = isDowncall(invokedMethod, classOfReceiver, mCall2@src,
              downcallCandidates, allInheritanceRels, extendsMap);
          if (downcallResult.downcallBool) {
            downcallLog += <<classOfReceiver, getDefiningClassOrInterfaceOfALoc(invokedMethod,
                invertedClassAndInterfaceContainment, projectM3 )>, <mCall2@src, invokedMethod, downcallResult.descDCallMeth>>;
          }
        }
      }
    }
  }
  for ( int i <- [0..size(downcallLog)]) {
    tuple [ inheritanceKey iKey, downcallDetail dDetail] aCase = downcallLog[i];
    resultRel += <aCase.iKey, DOWNCALL_ACTUAL>;
  }
  resultRel += {<<_child, _parent>, DOWNCALL_CANDIDATE> | <_parent, _child, _issMethod, _downcalledMethod> <- downcallCandidates
      };
  downcallLog += [<<_child, _parent>, <DEFAULT_LOC, _issMethod, _downcalledMethod>> | <_parent, _child, _issMethod,
      _downcalledMethod> <- downcallCandidates ];
  iprintToFile(getFilename(projectM3.id,downcallLogFile),downcallLog);
  return resultRel;
}
```

C.6 OtherUsesOfInheritance

C.6.1 Category Rascal Implementation from OtherInheritanceCases.rsc

```
private lrel [inheritanceKey, categorySibling] getOneCategoryCase(inheritanceKey anExtendsCandidate,  set [inheritanceKey]
    subtypeSet, map [loc, set[loc]] invertedDescMap) {
  lrel [inheritanceKey, categorySibling] retRel = [];
  bool isExtendsCategory = false;
  list [loc] allSiblings = toList(getAllDirectDescendants( anExtendsCandidate.parent, invertedDescMap));
  int i = 0;
  while ( (!isExtendsCategory)  && (i < (size(allSiblings)) ) ) {
    aSibling = allSiblings[i];
    inheritanceKey siblingKey = <aSibling, anExtendsCandidate.parent>;
    if (siblingKey in subtypeSet ) {
      isExtendsCategory = true;
      retRel += [<anExtendsCandidate, aSibling>];
    }
    i += 1;
  }
  return retRel;
}

public rel [inheritanceKey, inheritanceType] getCategoryCases(rel [inheritanceKey, inheritanceType] allInheritanceCases, M3
    projectM3) {
  rel [inheritanceKey, inheritanceType] retRel = {};
  lrel [inheritanceKey, categorySibling] categoryLog = [];
  map [loc, set[loc]] invertedExtendsMap =  getInvertedExtendsMap(projectM3);
  map [loc, set[loc]] invertedImplementsMap =  getInvertedImplementsMap(projectM3);

  set [loc] allSystemClasses = getAllClassesInProject(projectM3);
  set [loc] allSystemInterfaces = getAllInterfacesInProject(projectM3);
  set [inheritanceKey] allExplicitSystemCC = {<_child, _parent> | <_child, _parent> <- projectM3@extends, _child in
      allSystemClasses, _parent in allSystemClasses};
  set [inheritanceKey] allExplicitSystemCI = {<_child, _parent> | <_child, _parent> <- projectM3@implements, _child in
      allSystemClasses, _parent in allSystemInterfaces};
  set [inheritanceKey] allExplicitSystemII = {<_child, _parent> | <_child, _parent> <- projectM3@extends, _child in
      allSystemInterfaces, _parent in allSystemInterfaces};

  set [inheritanceKey] allUsedInheritanceRels = {<_child, _parent> | <<_child, _parent>, _iType> <- allInheritanceCases, _iType
      in {INTERNAL_REUSE, EXTERNAL_REUSE, SUBTYPE, DOWNCALL_ACTUAL, DOWNCALL_CANDIDATE, CONSTANT, MARKER, SUPER, GENERIC,
      FRAMEWORK} };
  set [inheritanceKey] subtypeSet = {<_child, _parent> | <<_child, _parent>, _iType> <- allInheritanceCases, _iType == SUBTYPE };

  set [inheritanceKey] CCCategoryCandidates = allExplicitSystemCC - allUsedInheritanceRels;
  set [inheritanceKey] CICategoryCandidates = allExplicitSystemCI - allUsedInheritanceRels;
  set [inheritanceKey] IICategoryCandidates = allExplicitSystemII - allUsedInheritanceRels;

  for (anExtendsCandidate <- (CCCategoryCandidates ) ) {
    categoryLog += getOneCategoryCase(anExtendsCandidate,  subtypeSet, invertedExtendsMap );
  }
  for (anImplementsCandidate <- CICategoryCandidates + IICategoryCandidates) {
    categoryLog += getOneCategoryCase(anImplementsCandidate,  subtypeSet, invertedImplementsMap);
  }

  iprintToFile(getFilename(projectM3.id,categoryLogFile), categoryLog);
  retRel = {<_iKey, CATEGORY> | <_iKey, _> <- categoryLog};
  return retRel;
}
```

C.6.2 Constant Rascal Implementation from OtherInheritanceCases.rsc

```

public bool areAllFieldsConstants(set [loc] fieldsInLoc, map[loc, set[Modifier]] allFieldModifiers) {
  bool retBool = false;
  for ( aField <- fieldsInLoc) {
    set [Modifier] modifiers = aField in allFieldModifiers ? allFieldModifiers[aField] : {} ;
    bool isFinal = false, isStatic = false, isPrivate = false;
    for (aModifier <- modifiers) {
      switch (aModifier) {
        case \private()  : isPrivate = true;
        case static()   : isStatic = true;
        case final()  : isFinal = true;
      } // switch
    }  // for
    if (!isPrivate && isFinal && isStatic) {
      retBool = true;
    }
    else {
      retBool = false;
      break;
    }
  } // for
  return retBool;
}


public bool containsOnlyConstantFields(loc aLoc, map[loc, set [loc]] classAndInterfaceContWithoutTypeVars, map[loc, set[Modifier
    ]] allFieldModifiers) {
  // we just want fields in the location, type variables are also added to be able to cover the generic constant classes
  bool retBool = false;
  set [loc] everythingInLoc = aLoc in classAndInterfaceContWithoutTypeVars ? classAndInterfaceContWithoutTypeVars[aLoc] : {} ;
  set [loc] fieldsInLoc = {_aField | _aField  <- everythingInLoc, isField(_aField)};
  bool allFieldsAreConstants = false;
  if (isInterface(aLoc)) {
    allFieldsAreConstants = true; // in an interface all fields are implicitly public static and final.
  }
  else {
    allFieldsAreConstants = areAllFieldsConstants(fieldsInLoc, allFieldModifiers);
  }
  if (!isEmpty(fieldsInLoc) && isEmpty(everythingInLoc - fieldsInLoc) && allFieldsAreConstants) {
    retBool = true;
  }
  return retBool;
}


public set [loc] getConstantCandidates(map[loc, set [loc]] classAndInterfaceContWithoutTypeVars, map[loc, set[Modifier]]
    allFieldModifiers, M3 projectM3) {
  set [loc] retSet = {};
  list [loc] allClassesAndInterfaces = sort(getAllClassesAndInterfacesInProject(projectM3));
  for (aLoc <- allClassesAndInterfaces ) {
    if (containsOnlyConstantFields(aLoc, classAndInterfaceContWithoutTypeVars, allFieldModifiers)) {
      retSet += aLoc;
    }
  }
  return retSet;
}


public bool areAllParentsInCandidateList(set [loc] allParentsOfLoc, set [loc] candidateLocs) {
  bool retBool = false;
  if (!isEmpty(allParentsOfLoc) && !isEmpty(candidateLocs) &&  (allParentsOfLoc <= candidateLocs) ) {
    retBool = true;
  }
  return retBool;
}


public rel [inheritanceKey,inheritanceType] getConstantLocs(set [loc] candidateLocs, M3 projectM3) {
  rel [loc, loc] allInheritanceRels = getNonFrameworkInheritanceRels(getInheritanceRelations(projectM3), projectM3);
  rel [inheritanceKey,inheritanceType] retRel = {};
  for (aLoc <- candidateLocs) {
    set [loc] allParentsOfLoc = {parent | <child, parent> <- allInheritanceRels, aLoc == child};
    if (isEmpty(allParentsOfLoc) || (areAllParentsInCandidateList(allParentsOfLoc, candidateLocs))) {
      retRel += {<<child, parent>, CONSTANT> | <child, parent> <- allInheritanceRels, aLoc == parent};
    }
  }
  return retRel;
}
```

C.6.3 Framework Rascal Implementation from OtherInheritanceCases.rsc

```
private rel [inheritanceKey, inheritanceType] getFrameworkCases(M3 projectM3) {
  rel [inheritanceKey, inheritanceType] retRel = {};
  rel [loc, loc]  allInheritanceRels  = getInheritanceRelations(projectM3);
  set [loc]     allTypesInProject   = carrier(allInheritanceRels);
  set [loc]     allSystemTypes    = getAllClassesAndInterfacesInProject(projectM3);
  set [loc]     allNonSystemTypes   = allTypesInProject - allSystemTypes;
  // we only want immediate parents
  rel [loc, loc]  extendsAndImplementsRel = {<_child, _parent> | <_child, _parent> <- projectM3@extends} + {<_child, _parent> | <
      _child, _parent> <- projectM3@implements} ;
  set [loc]     directChildrenOfNonSystemTypes = {_child | <_child, _parent> <- extendsAndImplementsRel, _parent in
      allNonSystemTypes};
  retRel = {<<_child, _parent>, FRAMEWORK> | <_child, _parent> <- extendsAndImplementsRel, _parent in
      directChildrenOfNonSystemTypes};
  return retRel;
}
```

C.6.4 Generic Rascal Implementation from OtherInheritanceCases.rsc

```

private rel [inheritanceKey iKey, set [loc] otherParents] getOneGenericUsage(Expression castStmt,   map [loc, set [loc]]
    invertedExtendsAndImplementsMap,
                                                    map [loc, set [loc]] extendsAndImplementsMap,
                                                    M3 projectM3 ) {
  rel [inheritanceKey iKey,set [loc] otherParents] oneUsage = {};
  visit (castStmt) {
    case \cast(castType, castExpr) : {
      TypeSymbol exprTypeSymbol = getTypeSymbolFromAnnotation(castExpr, projectM3); ;
      TypeSymbol castTypeSymbol = getTypeSymbolFromRascalType(castType);
      if (exprTypeSymbol := object()) {
        loc genericParentCandidate = getClassOrInterfaceFromTypeSymbol(castTypeSymbol);
        if (genericParentCandidate != DEFAULT_LOC) {
          set [loc] directDescendants = getAllDirectDescendants(genericParentCandidate , invertedExtendsAndImplementsMap );
          for (aDesc <- directDescendants) {
            set [loc] allParentsOfADesc = getAllDirectAscendants(aDesc, extendsAndImplementsMap);
            set [loc] otherParents = allParentsOfADesc - {genericParentCandidate};
            if (!isEmpty(otherParents)) {
              oneUsage += <<aDesc, genericParentCandidate>, otherParents>;
            }
          }
        }
      }
    }
  }
  return oneUsage;
}


private rel [inheritanceKey, inheritanceType] getGenericUsages(M3 projectM3) {
  rel [inheritanceKey, inheritanceType] retRel = {};
  lrel [inheritanceKey _iKey, set [loc] _otherParents, loc _castLoc] genericLog = [];
  map [loc, set [loc]] invertedExtendsAndImplementsMap  = getInvertedExtendsAndImplementsMap(projectM3);
  map [loc, set [loc]] extendsAndImplementsMap      = getExtendsAndImplementsMap(projectM3);
  set [Declaration] projectASTs = createAstsFromEclipseProject(projectM3.id, true);
  // search for methodCalls with passed parameter System type and declared parameter object(). Put all such system types to a set
      ,
  // because the otherParent we are looking for should also be in that set.
  for ( anAST <- projectASTs) {
    visit (anAST) {
      case castStmt:\cast(castType, castExpr) : {
        rel [inheritanceKey, set[loc]] oneGenericUsage = getOneGenericUsage(castStmt, invertedExtendsAndImplementsMap,
            extendsAndImplementsMap, projectM3 );
        if ( !isEmpty(oneGenericUsage) ) {
          tuple [inheritanceKey iKey, set [loc] otherParents] genericTuple = getOneFrom(oneGenericUsage);
          retRel += <genericTuple.iKey, GENERIC>;
          genericLog += <genericTuple.iKey, genericTuple.otherParents, castStmt@src>;
        }
      }
    }
  }
  iprintToFile(getFilename(projectM3.id,genericLogFile), genericLog);
  return retRel;
}
```

C.6.5 Marker Rascal Implementation from OtherInheritanceCases.rsc

```
public set [loc] getMarkerCandidates(map[loc, set[loc]] interfaceContainmentWithoutTypeVars, M3 projectM3) {
  set [loc] retSet = {};
  set [loc] allInterfaces = getAllInterfacesInProject(projectM3);
  set [loc] nonMarkerInterfaces = domain(interfaceContainmentWithoutTypeVars);
  retSet = allInterfaces - nonMarkerInterfaces;
  return retSet;
}


public rel [inheritanceKey, inheritanceType] getMarkerInterfaces(set [loc] markerCandidates, M3 projectM3) {
  rel [loc, loc] allInheritanceRels = getNonFrameworkInheritanceRels(getInheritanceRelations(projectM3), projectM3);
  rel [inheritanceKey,inheritanceType] retRel = {};
  for (anInterface <- markerCandidates) {
    set [loc] allParentsOfInterface = {parent | <child, parent> <- allInheritanceRels, anInterface == child};
    if (isEmpty(allParentsOfInterface) || (areAllParentsInCandidateList(allParentsOfInterface, markerCandidates))) {
      retRel += {<<child, parent>, MARKER> | <child, parent> <- allInheritanceRels, anInterface == parent};
    }
  }
  return retRel;
}
```

C.6.6 Super Rascal Implementation from OtherInheritanceCases.rsc

```
public rel [inheritanceKey, inheritanceType] getSuperRelations(M3 projectM3) {
  set [loc]         allClassesInProject     = getAllClassesInProject(projectM3);
  map [loc, set [loc]]  constructorContainmentMap   = toMap({<_owner, _constructor> | <_owner, _constructor> <-
      projectM3@containment, _constructor.scheme == "java+constructor" });
  map [loc, set [loc]]  extendsMap          = toMap({<_child, _parent> | <_child, _parent> <- projectM3@extends });
  map[loc, set[loc]]  implementsMap = toMap({<_child, _parent> | <_child, _parent> <- projectM3@implements });
  map [loc, set [loc]]  declarationsMap       = toMap({<aLoc, aProject> | <aLoc, aProject> <- projectM3@declarations});
  rel [inheritanceKey,inheritanceType] retRel = {};
  lrel [inheritanceKey, superCallLoc] superLog = [];
  for (aClass <- allClassesInProject) {
    set [loc] constructors = aClass in constructorContainmentMap ? constructorContainmentMap[aClass] : {} ;
    for (aConstructor <- constructors) {
      Declaration constructorAST = getMethodASTEclipse(aConstructor, model = projectM3);
      bool superCall = false; loc locOfSuperCall = DEFAULT_LOC;
      visit(constructorAST) {
        case s1:\constructorCall(isSuper:_,_,_) : {
          if (isSuper) {superCall = true; locOfSuperCall = s1@src;}
        }
        case s2:\constructorCall(isSuper:_,_) : {
          if (isSuper) {superCall = true; locOfSuperCall = s2@src;}
        }
      }
      if (superCall) {
        loc parentClass = getImmediateClassParentOfAClass(aClass, extendsMap, implementsMap, declarationsMap);
        retRel += <<aClass, parentClass>, SUPER>;
        superLog += <<aClass, parentClass>, locOfSuperCall>;
      }
    }
  }
  iprintToFile(getFilename(projectM3.id,superLogFile) ,superLog);
  return retRel;
}
```

C.7 InheritanceModules.rsc

InheritanceModules.rsc Rascal module contains many methods which are used by one or more modules described above. We do not include every module here, especially not the simple ones.

```

private set [loc] getInBetweenClasses(loc ascClass, loc descClass, map[loc, set[loc]] extendsMap) {
  set  [loc]  allInBetweenClasses = {};
  list [loc]  ascendantsInOrder = getAscendantsInOrder(descClass, extendsMap);
  bool ascClassFound = false;
  int i = 0;
  while ((!ascClassFound) && (i < size(ascendantsInOrder))) {
    loc anAscendant = ascendantsInOrder[i];
    if (anAscendant == ascClass) {
      ascClassFound = true;
    }
    else {
      allInBetweenClasses += anAscendant;
    }
    i += 1;
  }
  if (!ascClassFound) { throw "Ascending␣class␣<ascClass>␣is␣not␣found␣in␣the␣Ascendants␣list␣of␣class:␣<descClass>"; }
  return allInBetweenClasses;
}


public bool isMethodOverriddenByAnyDescClass(loc aMethod, loc ascClass, loc descClass, map[loc, set[loc]] invertedContainment, map[loc, set[loc]] extendsMap, M3 projectM3) {
  bool retBool = false;
  set  [loc]  classesThatOverrideTheMethod = getClassesWhichOverrideAMethod(aMethod, invertedContainment, projectM3);
  set  [loc]  allDescClassesUpToDescClass =  descClass + getInBetweenClasses(ascClass, descClass, extendsMap);
  if ( !isEmpty(classesThatOverrideTheMethod & allDescClassesUpToDescClass) ) {
    retBool = true;
  }
  return retBool;
}


private loc getImmediateParentOfInterface(loc classOrInt, map[loc, set[loc]] extendsMap, map[loc, set[loc]] declarationsMap  ) {
  loc retLoc    = DEFAULT_LOC;
  set [loc] immediateParentInterfaceSet   = classOrInt in extendsMap ? extendsMap[classOrInt] : {};
  if (size(immediateParentInterfaceSet) == 1) {
    retLoc = getOneFrom(immediateParentInterfaceSet);
  }
  else {
    if (isEmpty(immediateParentInterfaceSet)) {
    ;}
    else {
      set [loc] immediateParentsInSystem = {_parent | _parent <- immediateParentInterfaceSet, isLocDefinedInProject(_parent, declarationsMap) };
      if (isEmpty(immediateParentsInSystem)) { retLoc = getOneFrom(immediateParentInterfaceSet); }
      else {retLoc = getOneFrom(immediateParentsInSystem);}
    }
  }
  return retLoc;
}



```

```
public loc getImmediateParentOfClass(loc classOrInt, map[loc, set[loc]] extendsMap,   map[loc, set[loc]] implementsMap, map[loc, set[loc]] declarationsMap) {
  loc retLoc    = DEFAULT_LOC;
  set [loc] immediateParentClassSet     = classOrInt in extendsMap ? extendsMap[classOrInt] : {};
  set [loc] immediateParentInterfaceSet   = classOrInt in implementsMap ? implementsMap[classOrInt] : {};
  if (immediateParentClassSet != {}) {
    if (size(immediateParentClassSet) >1) { throw "in␣getImmediateParentOfClass,␣loc␣<classOrInt>,␣has␣more␣than␣one␣class␣parents:␣<immediateParentClassSet>";}
    retLoc = getOneFrom(immediateParentClassSet);
  }
  else {
    if (size(immediateParentInterfaceSet) == 1) {
      retLoc = getOneFrom(immediateParentInterfaceSet);
    }
    else {
      if (isEmpty(immediateParentInterfaceSet)) {
      ;}
      else {
        set [loc] immediateParentsInSystem = {_parent | _parent <- immediateParentInterfaceSet, isLocDefinedInProject(_parent, declarationsMap) };
        if (isEmpty(immediateParentsInSystem)) { retLoc = getOneFrom(immediateParentInterfaceSet); }
        else {retLoc = getOneFrom(immediateParentsInSystem);}
      }
    }
  }
  return retLoc;
}





public loc getImmediateParent (loc classOrInt, map[loc, set[loc]] extendsMap, map[loc, set[loc]] implementsMap, map[loc, set[loc]] declarationsMap  ) {
  if (isClass(classOrInt)) {return getImmediateParentOfClass(classOrInt, extendsMap, implementsMap, declarationsMap); }
  if (isInterface(classOrInt)) {return getImmediateParentOfInterface(classOrInt, extendsMap, declarationsMap); }
  println("classOrInt␣is␣neither␣class␣nor␣interface:␣<classOrInt>" );
  return DEFAULT_LOC;
}




private loc getImmediateParentGivenAnAscForCC_II(loc classOrInt, loc ascLoc,  map[loc, set[loc]] extendsMap, rel [loc, loc] allInheritanceRelations) {
  loc retLoc    = DEFAULT_LOC;
  loc foundLoc  = DEFAULT_LOC;
  set [loc] immediateParentInterfaceSet   = classOrInt in extendsMap ? extendsMap[classOrInt] : {};
  if (size(immediateParentInterfaceSet) == 1) {
    retLoc = getOneFrom(immediateParentInterfaceSet);
  }
  else {
    for (anImmediateParent <- immediateParentInterfaceSet) {
      if (inheritanceRelationExists(anImmediateParent, ascLoc, allInheritanceRelations)) {
        foundLoc = anImmediateParent;
        break;
      }
    } // for
  retLoc = foundLoc;
  }
  return retLoc;
}







private loc getImmediateParentGivenAnAscForCI(loc classOrInt, loc ascLoc,  map[loc, set[loc]] extendsMap,   map[loc, set[loc]] implementsMap,
                                                          rel [loc, loc] allInheritanceRelations) {
  loc retLoc    = DEFAULT_LOC;
  loc foundLoc  = DEFAULT_LOC;
  bool interfaceParentFound = false;
  set [loc] immediateParentClassSet     = classOrInt in extendsMap    ? extendsMap[classOrInt] : {};
  set [loc] immediateParentInterfaceSet   = classOrInt in implementsMap   ? implementsMap[classOrInt] : {};
  if (ascLoc in immediateParentInterfaceSet) {
    retLoc = ascLoc;
    interfaceParentFound = true;
  }
  else {
    for (anImmediateParent <- immediateParentInterfaceSet) {
      if (inheritanceRelationExists(anImmediateParent, ascLoc, allInheritanceRelations)) {
        foundLoc = anImmediateParent;
        interfaceParentFound = true;
        break;
      }
    } // for
    retLoc = foundLoc;
  }
  if (!interfaceParentFound) {
    if (immediateParentClassSet != {}) {
      if (size(immediateParentClassSet) > 1) { throw "in␣getImmediateParentOfClassGivenAnAsc,␣loc␣<classOrInt>,␣has␣more␣than␣one␣class␣parents:␣<immediateParentClassSet>";}
      retLoc = getOneFrom(immediateParentClassSet);
    }
  }
  return retLoc;
}




public loc getImmediateParentGivenAnAsc(loc classOrInt, loc ascLoc,  map[loc, set[loc]] extendsMap, map[loc, set[loc]] implementsMap,  rel [loc, loc] allInheritanceRelations) {
  if (isClass(classOrInt)) {
    if (isClass(ascLoc)) { return getImmediateParentGivenAnAscForCC_II(classOrInt, ascLoc,  extendsMap, allInheritanceRelations); }
    if (isInterface(ascLoc)) { return getImmediateParentGivenAnAscForCI(classOrInt, ascLoc,  extendsMap, implementsMap, allInheritanceRelations); }
    else {
      println("The␣loc␣<ascLoc>␣is␣not␣a␣class␣or␣interface.");
      return DEFAULT_LOC;
    }
  }
  if (isInterface(classOrInt)) { return getImmediateParentGivenAnAscForCC_II(classOrInt, ascLoc,  extendsMap, allInheritanceRelations); }
  println("The␣loc␣<classOrInt>␣is␣not␣a␣class␣or␣interface.");
  return DEFAULT_LOC;
}



public lrel [loc, loc] getInheritanceChainGivenAsc(loc classOrInt, loc ascLoc,  map[loc, set[loc]] extendsMap,  map[loc, set[loc]]  implementsMap,
                                                        map[loc, set[loc]]  declarationsMap,
                                                        rel [loc, loc]    allInheritanceRelations) {
  lrel [loc, loc] retList = [];
  bool topReached = false;
  loc childType = classOrInt;
  loc immediateParent = getImmediateParentGivenAnAsc(classOrInt, ascLoc, extendsMap, implementsMap, allInheritanceRelations);
  if (ascLoc != DEFAULT_LOC) {
    while (!topReached) {
      if (immediateParent == DEFAULT_LOC) {
        topReached = true;
      }
      else {
        retList = retList + <childType, immediateParent>;
        if ((immediateParent == ascLoc) || !(isLocDefinedInProject(immediateParent, declarationsMap)) ) { topReached = true;}
        else {
          childType = immediateParent;
          immediateParent= getImmediateParentGivenAnAsc(childType, ascLoc, extendsMap, implementsMap, allInheritanceRelations);
        }
      }
    }
  }
  return retList;
}

public tuple [bool, inheritanceKey] getSubtypeRelation(TypeSymbol childSymbol, TypeSymbol parentSymbol) {
  bool isSubtypeRel = false;
  inheritanceKey iKey = <DEFAULT_LOC, DEFAULT_LOC>;
  iKey.parent = getClassOrInterfaceFromTypeSymbol(parentSymbol);
  iKey.child = getClassOrInterfaceFromTypeSymbol(childSymbol);
  if ((iKey.child != DEFAULT_LOC) && (iKey.parent != DEFAULT_LOC) && (iKey.child != iKey.parent)) {
    isSubtypeRel = true;
  }
  return <isSubtypeRel, iKey>;
}


loc getTypeVariableFromTypeSymbolForClassOrInt(TypeSymbol aTypeSymbol) {
  loc typeVar = DEFAULT_LOC;
  visit (aTypeSymbol) {
    case _typeArgument:\typeParameter(loc decl, _) : {
      typeVar = decl;
    }
  }
  if (typeVar == DEFAULT_LOC)  { throw "No␣type␣variable␣is␣found␣for␣<aTypeSymbol>"; }
  return typeVar;
}




loc getTypeVariableFromTypeSymbol(TypeSymbol aTypeSymbol) {
  loc typeVar = DEFAULT_LOC;
  visit (aTypeSymbol) {
    case _typeArgument:\typeArgument(loc decl) : {
      typeVar = decl;
    }
  }
  if (typeVar == DEFAULT_LOC)  { throw "No␣type␣variable␣is␣found␣for␣<aTypeSymbol>"; }
  return typeVar;
}


list [loc] getTypeVariablesOfRecClass(loc recClassOrInt, map [loc, set [TypeSymbol]] typesMap) {
  TypeSymbol typeSymbolOfLoc = getTypeSymbolOfLocDeclaration(recClassOrInt, typesMap);
  list [TypeSymbol] typeSymbolParList = [];
  list [loc] typeVariablesList = [];
  visit (typeSymbolOfLoc) {
    case aClass:\class(loc decl, list[TypeSymbol] typeParameters) : {
      typeSymbolParList = typeParameters;
    }
    case anInterface:\interface(loc decl, list[TypeSymbol] typeParameters) : {
      typeSymbolParList = typeParameters;
    }
  }
  for (aTypePar <- typeSymbolParList) {
    typeVariablesList += getTypeVariableFromTypeSymbolForClassOrInt(aTypePar);
  }
  return typeVariablesList ;
}


map [loc, TypeSymbol] getTypeVariableMap(list [loc] typeVariables, list [TypeSymbol] typeParameters, loc stmtLoc, M3 projectM3) {
  map [loc, TypeSymbol] retMap = ();
  list [TypeSymbol] completeTypeParameters = typeParameters;
  if (size(typeVariables) != size(typeParameters)) {
    if (size(typeVariables) > size(typeParameters)) {
      // this is OK, Java allows instantiation with absent type parameters, we will fill the rest of the typeParameters with Object type symbol
      int numOfElementsToAdd = size(typeVariables) - size(typeParameters);
      for (int i <- [0..numOfElementsToAdd]) {
        completeTypeParameters += OBJECT_TYPE_SYMBOL;
      }
      println("Less␣type␣parameters␣than␣type␣variables!␣Type␣variables:␣<typeVariables>,␣type␣parameters:␣<typeParameters>␣at:␣<stmtLoc>");
      ;
    }
    else {
      // exceptional situation, we will log this in error log and return an empty map.
      println("More␣type␣parameters␣than␣type␣variables!␣Type␣variables:␣<typeVariables>,␣type␣parameters:␣<typeParameters>␣at:␣<stmtLoc>");
      appendToFile(getFilename(projectM3.id, errorLog), "More␣type␣parameters␣than␣type␣variables!␣Type␣variables:␣<typeVariables>,␣type␣parameters:␣<typeParameters>␣at:␣<stmtLoc>\n\n");
      completeTypeParameters = [];
    }
  }
  for (int i <- [0..size(completeTypeParameters)] ) {
    retMap += (typeVariables[i] : completeTypeParameters[i]);
  }
  return retMap;
}


list [TypeSymbol]  getReceivingTypeParameters(TypeSymbol recTypeSymbol) {
  list [TypeSymbol] recTypeParameters = [];
  visit (recTypeSymbol) {
    case classDef:\class(loc decl, list[TypeSymbol] typeParameters) : {
      recTypeParameters = typeParameters;
    }
    case intDef:\interface(loc decl, list[TypeSymbol] typeParameters) : {
      recTypeParameters = typeParameters;
    }
  }
  if (isEmpty(recTypeParameters)) {
    // in case of a class which is declared with type parameter(s), but still is instantiated without
    // type parameters, the recTypParameters can be empty. I handle this further in the calling method.
    // throw "Receiver type parameters is empty for receiver : <recTypeSymbol>";
  ;}
  return recTypeParameters;
}



TypeSymbol resolveGenericTypeSymbol(TypeSymbol genericTypeSymbol, Expression methodOrConstExpr, map [loc, set [TypeSymbol]] typesMap,
                                  map[loc, set[loc]] invertedClassAndInterfaceContainment, M3 projectM3 ) {
  loc methodParameterTypeVariable = getTypeVariableFromTypeSymbol(genericTypeSymbol);
  TypeSymbol resolvedTypeSymbol = genericTypeSymbol;
  TypeSymbol recTypeSymbol = DEFAULT_TYPE_SYMBOL;
  loc methodOwningClassOrInt = DEFAULT_LOC;
  switch (methodOrConstExpr) {
    case mCall:\methodCall(_,receiver:_,_,_) : {
      methodOwningClassOrInt =  getDefiningClassOrInterfaceOfALoc(methodOrConstExpr@decl, invertedClassAndInterfaceContainment, projectM3);
      if (methodOwningClassOrInt != DEFAULT_LOC) {
        recTypeSymbol = receiver@typ;
      }
    }
    case mCall:methodCall(_,_,_) : {
      // There can be no subtyping between type parameters, so I do not have to do anything here.
    ;}
    case newObject1:\newObject(Type \type, list[Expression] expArgs) : {
      recTypeSymbol =  getTypeSymbolFromRascalType(\type);
      methodOwningClassOrInt =  getClassOrInterfaceFromTypeSymbol(recTypeSymbol);
    }
    case newObject2:\newObject(Type \type, list[Expression] expArgs, Declaration class) : {
      recTypeSymbol =  getTypeSymbolFromRascalType(\type);
      methodOwningClassOrInt =  getClassOrInterfaceFromTypeSymbol(recTypeSymbol);
    }
    case newObject3:\newObject(Expression expr, Type \type, list[Expression] expArgs) : {
      recTypeSymbol =  getTypeSymbolFromRascalType(\type);
      methodOwningClassOrInt =  getClassOrInterfaceFromTypeSymbol(recTypeSymbol);
    }
    case newObject4:\newObject(Expression expr, Type \type, list[Expression] expArgs, Declaration class) : {
      recTypeSymbol =  getTypeSymbolFromRascalType(\type);
      methodOwningClassOrInt =  getClassOrInterfaceFromTypeSymbol(recTypeSymbol);
    }
  }
  if (recTypeSymbol != DEFAULT_TYPE_SYMBOL) {
    // recTypeParameters holds the actual types with which the object was instantiated, like Shape, Ractangle, String, etc.
    list [TypeSymbol] recTypeParameters = getReceivingTypeParameters(recTypeSymbol);
    if ( !isEmpty(recTypeParameters) ) {
      // typeVariablesOfRecClass holds the type variables in the class definition, like X, T in <X,T>
      list  [loc] typeVariablesOfRecClass       = getTypeVariablesOfRecClass(methodOwningClassOrInt, typesMap); // type variables
           like T, X
      // typeVariableMap holds the pair (typeVariable : typeParameter) with respect to the object, like (X : Shape)
      map   [loc, TypeSymbol] typeVariableMap     = getTypeVariableMap(typeVariablesOfRecClass, recTypeParameters,
          methodOrConstExpr@src, projectM3);
      if (methodParameterTypeVariable notin typeVariableMap) {
        println("methodParameterTypeVariable:␣<methodParameterTypeVariable>␣is␣not␣found␣in␣typeVariableMap:␣<typeVariableMap>");
        println("method␣call␣to␣method:␣<methodOrConstExpr@decl>␣at␣source␣location:␣<methodOrConstExpr@src>␣");
        resolvedTypeSymbol = DEFAULT_TYPE_SYMBOL;
      }
      else {
        resolvedTypeSymbol = typeVariableMap[methodParameterTypeVariable];
      }
    }
    else {
      resolvedTypeSymbol  = OBJECT_TYPE_SYMBOL;
    }
  }
  return resolvedTypeSymbol;
}




public list [TypeSymbol] updateTypesWithGenerics(Expression methodOrConstExpr, list [TypeSymbol] typeList, map [loc, set[
    TypeSymbol]] typesMap,
                                        map[loc, set[loc]] invertedClassAndInterfaceContainment, M3 projectM3  ) {
  list [TypeSymbol] retList = [];
  TypeSymbol currentTypeSymbol = DEFAULT_TYPE_SYMBOL;
  for (_aTypeSymbol <- typeList) {
    currentTypeSymbol = _aTypeSymbol;
    visit (_aTypeSymbol) {
      case aTypeArg:\typeArgument(loc decl) : {
        currentTypeSymbol = resolveGenericTypeSymbol(_aTypeSymbol, methodOrConstExpr, typesMap,
            invertedClassAndInterfaceContainment, projectM3);
      }
    }
    retList += currentTypeSymbol;
  }
  return retList;
}


public list [TypeSymbol] getDeclaredParameterTypes (Expression methodOrConstExpr, map [loc, set[TypeSymbol]] typesMap, map[loc,
    set[loc]] invertedClassAndInterfaceContainment,
                                                            M3 projectM3 ) {
  list [TypeSymbol] retTypeList   = [];
  list [TypeSymbol] methodParameterTypes = [];
  loc methodLoc           = methodOrConstExpr@decl;
  TypeSymbol methodTypeSymbol = getTypeSymbolOfLocDeclaration(methodLoc, typesMap);
  visit (methodTypeSymbol) {
    case \method(_, _, _, typeParameters:_) : {
      methodParameterTypes = typeParameters;
    }
    case cons:\constructor(_, typeParameters:_) : {
      methodParameterTypes = typeParameters;
    }
  }
  retTypeList = updateTypesWithGenerics(methodOrConstExpr, methodParameterTypes, typesMap, invertedClassAndInterfaceContainment,
      projectM3);
  return retTypeList;
}


public TypeSymbol getDeclaredReturnTypeSymbolOfMethod(loc methodLoc, map [loc, set[TypeSymbol]] typesMap ) {
  TypeSymbol retSymbol = DEFAULT_TYPE_SYMBOL;
  TypeSymbol methodTypeSymbol = getTypeSymbolOfLocDeclaration(methodLoc, typesMap);
  visit (methodTypeSymbol) {
    case \method(_, _, returnType,  _) : {
      retSymbol = returnType;
    }
  } // visit
  return retSymbol;
}


public Expression createMethodCallFromConsCall(Statement consCall) {
  Expression retExp;
  list [Expression] arguments = [];
  visit (consCall) {
       case consCall1:\constructorCall(_, args:_): {
      arguments = args;
        }
        case consCall2:\constructorCall(_, expr:_, args:_): {
      arguments = args;
        }
  }
  retExp = methodCall(true, "␣", arguments);
  retExp@decl = consCall@decl;
  retExp@src  = consCall@src;
  return retExp;
}
```