Master Thesis Title

Cigdem Aytekin
cigdem.aytekin2@student.uva.nl

June 12, 2014, 27 pages

Supervisor: Tijs van der Storm
Host organisation: CWI - Centrum voor Wiskunde en 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 classes. 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. 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. Moreover, they found out that downcall (late-bound self-reference) was also used quite frequently, about a third of inheritance relationships included downcall 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. using Rascal meta-programming language. We use the inheritance model of the original study and also the open source projects from Qualitas Corpus, but we analyse the Java source code instead of the byte code. Our study is still in progress and therefore we do not report any results at the moment.

Chapter 1

Problem Statement and Motivation

This chapter will provide an introduction. TODO !!!!

1.1 Java Inheritance

TODO!!!!

1.2 The Reasons for Conducting a Replication

TODO!!!!

Chapter 2

The Original Study

2.1 Overview

Ewan Tempero, Hong Yul Yang and James Noble have published the article "What Programmers Do With Java?" in 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, 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 therefore it deserves a chapter of its own (ref. xxxxxxx to definition chapter).

Tempero et al. conducted research about Java inheritance before this study as well. Their previous inheritance research concentrated on the existing inheritance relationships in Java projects [reference: xxxxxx]. This study is different in the sense that they wanted to find out for which purposes inheritance was actually used in Java this time. In their 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 P and Q. If Q is defined as a subclass of P, then we talk about an inheritance relationship between P and Q. If, however, a method is defined in P and is not overridden in Q (say, method p()) is actually called on an object of type Q, we talk about an inheritance usage because then we see that 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 which represents the usage of Java inheritance. Secondly, they make their data sets and their results available for other research. Finally, they present their study results which imply that inheritance is used quite considerably in open source Java projects, especially for subtyping and what they call external reuse.

2.2 Research Questions

The following are the research questions of the original study (They are named ORQ (Original Research Question) to distinguish them from the research questions of the replication study.):

ORQ1:	To what extent is late-bound self-reference relied on in the designs of Java Systems? (The terms late-bound self-reference and downcall are synonyms in the study and are defined in [xxxxxxxxxxxxxx])
ORQ2:	To what extent is inheritance used in Java in order to express a subtype relationship that is necessary to design?
ORQ3:	To what extent can inheritance be replaced by composition?
ORQ4:	What other inheritance idioms are in common use in Java systems?

2.3 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: ant (20 releases in total, from release 1.1 to 1.8.1) and freecol (23 releases in total, from 0.3.0 to 0.9.4). Corpus website is: [TAD+08]

The Corpus consists of both byte codes and source codes of the projects.

Qualitas.class Corpus Some effort is needed to create the compiled versions of Java source code from Qualitas Corpus. Terra et al. conducted a study [TMVB13b] and made their results available in the website: [TMVB13a]

This is not the original part of the study and should be moved to the replication part TODO !!!!!!!!!!!!!!!

Study Results Web Page In addition to the results documented in their article, authors also have placed a package of detailed information on a web site [TYN08]. They document here the definitions, edge attributes and metrics they have used for the study as well as the results of their measurements on the Corpus projects. The [definitions chapter] documents this information in detail.

Communication with the Authors 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 clear after mailing with the authors. The e-mails are available

This is not the original part of the study and should be moved to the replication part TODO!!!!!!!!!!!!!!!

2.4 Limitations of the Original Study

The limitations of the original study are as follows:

- The study is limited to Java classes and interfaces, exceptions, enums and annotations are excluded,
- The third party libraries are not analysed,
- The edges between system types and non-system types are not modelled,
- Heuristics are used when defining framework and generics attributes,
- The authors use the Java byte code as input to their analysis tool, byte code may in some cases incorrectly map to source code,
- They do make static code analysis and this may have impact on their down call results, the results may be overstating the reality

2.5 Analysis Challenges of the Original Study

The authors report three challenges:

- They do not anlayze the third party libraries and therefore they can not determine the purpose of some inheritance relationships. They suspect in some instances that two types may have subtype relationship, but they can not say for sure because of this limitation. They have introduced the framework 3.9.3 attribute to cover such cases.

- The definition of generic 3.9.4 is also introduced to cover the cases in which they suspect of a subtype reuse.
- Byte code may not always map on to the source code correctly. This happens rarely, and it may cause some false negatives, for example in external reuse cases, during the analysis.

2.6 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 edges involve down calls.

For Research Question 2: At least two thirds 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 introduces opportunities to replace 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

Chapter 3

Definitions

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

The authors of the original study model inheritance relationships in a graph. The descendant ascendant types are modelled as edges of the graph and 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 reasons for this choice are discussed in the [xxxxxxxxxx] section.

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*.

3.1 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 is not defined in the system. In the example, the class G is a system type and ArrayList is a non-system type.

```
import java.util.ArrayList;

public class G extends ArrayList { }
```

3.2 User defined attribute

The descendant ascendant pair in an inheritance relationship has user defined attribute if both of 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;
```

3.3 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 {}
```

3.4 Explicit attribute

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

```
class P{ }
class C extends P { }
class G extends C {  }
```

3.5 Internal Reuse

Internal reuse happens when a descendant type calls a method or accesses a field of its ascendant type.

```
public class P {
    public int pField = 0;
    void p() {
    }
}
public class Q extends P {
    void q() {
        p();         // via method call
        pField = 1;  // via field access
    }
}
```

3.6 External Reuse

External reuse is like internal reuse, except for that the access to a method or a field happens not within the descendant type itself, but it happens in another type, 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.

```
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();          // via method call
        aQ.pField = 1;   // via field access
    }
}
```

3.7 Subtype

Subtype usage happens when an object of descendant type is supplied where an object of ascendant type is expected. 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 enhanced for loop in Java is also like an assignment statement and therefore should be included in the analysis.

There are two interesting cases of subtyping usage in Java (sideways cast and this changing type) and they are defined separately in the subsections below.

```
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
            // ...
        }
    }
}
```

3.7.1 Sideways Casting

Sideways casting is an interesting case which results in subtype usage between a class and two interfaces. The example is taken as is from the original study.

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

3.7.2 This Changing Type

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

3.8 Downcall

The terms downcall and late-bound self-reference have the same meaning in the original study. Downcall 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()
}
```

3.9 Other Uses of Inheritance

Next to reuse, subtype and downcall, the authors also defined other uses of inheritance: Category, Constants, Framework, Generic, Marker and Super.

3.9.1 Category

Category inheritance relationship is defined for the descendant ascendant pairs which can not be placed under any other inheritance definition. (We should also note that for this definition, ascendant type should be direct ascendant of the descendant type, i.e. no types are 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() {
     // subtype usage btw. S and P
        P aP = new S();
    }
}
```

3.9.2 Constants

A descendant ascendant pair has constants attribute if the ascendant only contains constant fields (i.e., fields with `static final` attribute). The ascendant should either have no ascendant it self or if it has ascendants, the pair ascendant-(grand)ascendant should also have constants attribute. In the example, B A pair 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 { }
```

3.9.3 Framework

A descendant-ascendant pair will have the framework attribute if it does not have one of the external reuse, internal reuse, subtype or downcall attributes and the ascendant is a direct descendant of a third party type. Moreover, the first type should be direct descendant of the second type. In the example, H G pair has Framewrok attribute.

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

public class H extends G { }
```

3.9.4 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

I again take the example from the original study.

```
List list = new Vector();
T aT = new R();
list.add(aT);
S anS = (S)list.get(0);
```

3.9.5 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, ascendant-(grand)ascendant pairs should all have marker attribute. 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() { }
}
```

3.9.6 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();
    }
}
```

3.10 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. Downcall is only defined for CC relations, whereas the ascendant type in the marker usage can only be an interface. The table 3.1 lists the possible combinations:

In the scope of the replication study: I do not implement Internal Reuse CI and II, Subtype-This Changing Type CI and Category II. Outside of scope !!!!!!!!!!!!!!!!!!! Do not forget to write that down.

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

Table 3.1: Possible inheritance usages in class-class, class-interface and interface-interface pairs.

Chapter 4

Metrics

The metrics are explained elaborately in the website of the original study [TYN08] The classifications used in the published article are based on these metrics. For example, Figure 12 of the article depicts different uses of inheritance on CC edges. The abbreviations INO, EX-ST and ST do correspond to three CC metrics respectively (numCCUsedOnlyInRe, numCCExreuseNoSubtype, perCCSubtype). Although most of the time it was possible to infer these correspondences from the abbreviations, we also e-mailed the authors to acquire the exact relationship between the metrics and the abbreviations. These correspondences are explained in section 4.4

4.1 Class Class (CC) Metrics

The metrics about CC (Class Class) inheritance relations are explained in table 4.1. For all metrics it holds that the descendant ascendant pair should have explicit and user defined attributes.

`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 downcall use was seen. `numCCDC / numCCUsed`
`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 attributes defined.
	`numCCUnexplCategory / numExplicitCC`

Table 4.1: Class Class Metrics

4.2 Class Interface (CI) Metrics

Table 4.2 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 use but do have some other attribute (one of 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 attributes defined. numUnexplainedCI/ numExplicitCI

Table 4.2: Class Interface Metrics

4.3 Interface Interface (II) Metrics

II (Interface Interface) metrics are depicted in table 4.3. 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
numExplainedII perExplainedII	Number of II edges which do not have subtype 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 4.3: Interface Interface Metrics

4.4 Correspondence between the metrics and article results

The correspondence between the inheritance metrics used in the study and the abbreviations used in the published article are shown in table 4.4

In article figures:	Term in the article	Name of the metric
Fig. 10 and 11 CC Downcalls	Downcall 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 4.4: Correspondence between the terms used in the article and the metrics used in the study.

Chapter 5

Replication Study

Our study has two objectives: firstly, to replicate the original study (replication) and secondly, to find out if a correlation between the project size and inheritance usage exists (extension). The same inheritance usage model from the original article is used in the second part of the study. Most of the time and effort is invested in the replication study.

In this chapter, the replication study is explained in detail. Research questions are introduced first, followed by an explanation of the study set-up, especially how it differs from the original study. Then we explain the challenges we faced during the replication study, we faced with some different challenges, mainly because we analyse the source code, and Tempero et al. analyse the byte code. Their challenges were discussed in section 2.5. Finally the results of the replication study are reported.

We found it useful to add another chapter for solely discussing the difference between two studies. Such a comparison will inevitably contain information which is already given previously. It is, however, very useful to bring information from the two studies together side by side, because it is a very comprehensible way of discussing the differences between two studies.

5.1 Research Questions

The research questions for the replication are based on the results of the original study which were discussed in section 2.6. 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.:

RQ1:	How do our results differ from the original study in downcalls?
RQ2:	How do our results differ from the original study in subtype inheritance usage?
RQ3:	How do our results differ from the original study in external reuse?
RQ4:	How do our results differ from the original study in other uses of inheritance?

Our research questions for the extension will be about the size of the project and different usages of inheritance. Just like the previous four questions, they are based on the four research questions of the original article 2.2

RQ5:	Is there a correlation between the size of the project and the number of descendant ascendant type pairs which use downcall?
RQ6:	Is there a correlation between the size of the project and the number of descendant ascendant type pairs which have subtype relationship?
RQ7:	Is there a correlation between the size of the project and the number of descendant ascendant type pairs which make use of external reuse?
RQ8:	Is there a correlation between the size of the project and other uses of inheritance?

5.2 Differences in the Study Set-up

Source code versus byte code: The biggest difference between the original and replication studies is about the input to analysis work. 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 Analysis Challenges section. 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 and is described in detail in the section 5.3.

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 paragraph.

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 form and source form. Authors explain in detail how they have obtained the byte and source forms 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, however, is not validated. It may very well be possible that different set of classes or interfaces are included in binary and source forms. One may rightfully expect that many classes and interfaces would be included in both of the forms, however, there may be differences between the two.

We expect that this difference may cause the biggest differences in the results, since the set of analysed types will be different in the original study and in the replication study.

Qualitas Corpus vs. Qualitas.class Corpus: The byte codes 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. However, acquiring the source code is not always enough for a successful compilation of the code. We are using the meta-programming language Rascal, and our Rascal program requires that the source is successfully compiled. As explained by Terra et al. in [TMVB13b], sometimes there is additional work needed for a successful compilation: for example, the dependencies of the Java projects need to be resolved. Terra et al. already studied on this and made a compiled version of the Qualitas Corpus: Qualitas.class Corpus. The compiled projects are available from their website [TMVB13a].

The original study used the 20101126 release of the Qualitas Corpus and 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. The list of projects (with the versions) we use in our analysis, as well as the list of version differences of 27 projects can be found in .1.

The different versions of a project will inevitably contain different code. This is likely to 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.

5.3 Analysis Challenges of the Replication Study

Our study is based on the original study in almost all aspects. Therefore the challenges they have faced 2.5 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 3.9.3 and generic 3.9.4) 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 section we first explain the generics in Java. We then briefly describe the concept of type erasure. We finally demonstrate the challenge we faced during the type analysis of generics.

5.3.1 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. 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> { }

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 extends P>` is the *type parameter* for class `SubArrayList`. `T` is called the *type variable*. `SubArrayList` is called a *parametrized 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>`.

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.

5.3.2 Type Erasure

During compilation, Java compiler replaces the type variables with supplied type parameters. This mechanism is called *type erasure*. Type erasure is already applied in the byte code. We analyse source code. It is very useful to list differences between the two, since the authors and we are actually analysing different code when it comes to generic types. We will continue to use our example from 5.3.1. After 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     { }

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 of `SubArrayList`:

  ```
  public class SubArrayList            extends ArrayList     { }
  ```

- 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);
  ```

5.3.3 Type Analysis of Generics

As mentioned earlier, we are using Rascal meta programming language for analysis. When we analyse a Java statement, in most cases, we can obtain the type of an expression easily. 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> {
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()`, 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.

This presents the most important challenge we faced during the code analysis. We do not think that our analysis results will contain differences due to this challenge, because we do apply the procedure as type erasure in our approach.

5.4 Limitations and Scope

5.5 Results of the Replication Study

Chapter 6

Comparison of Original Study with Replication Study

6.1 Comparison of Results

Chapter 7

Discussion

7.1 Limitations of the Study

7.2 Threats to Validity

7.3 Lessons Learned

7.4 Future Work

Chapter 8

Conclusion

This is my thesis.

Bibliography

[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.

[TMVB13a] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. Qualitas.class Corpus. 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 Corpusxxx. *Software Engineering Notes*, 38(5):1–4, 2013. doi:10.1145/2507288.2507314.

[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.

.1 Appendix A - Analysed Projects

The following table lists the projects analysed in the replication study.

ant-1.8.2 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 drjava-stable-20100913-r5387 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 jre-1.6.0 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 roller-5.0.1 rssowl-2.0.5 sablecc-3.2 springframework-3.0.5 squirrel_sql-3.1.2 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 1: List of analysed projects in the replication study

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

Original Study	Replication Study
ant-1.8.1	ant-1.8.2
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-beta4	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
jre-1.5.0_22	jre-1.6.0
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
roller-4.0.1	roller-5.0.1
sablecc-3.1	sablecc-3.2
springframework-1.2.7	springframework-3.0.5
weka-3.7.2	weka-3.6.9

Table 2: Versions of the projects which are different between the original study and replication study.