Master Thesis Title

Cigdem Aytekin
cigdem.aytekin2@student.uva.nl

June 5, 2014, 21 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

This is the abstract

Chapter 1

Problem Statement and Motivation

This chapter will provide an introduction.

1.1 Java Inheritance

1.2 The Reasons for Conducting a Replication

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:

Research Question 1 (RQ1): 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])

Research Question 2 (RQ2): To what extent is inheritance used in Java in order to express a subtype relationship that is necessary to design?

Research Question 3 (RQ3): To what extent can inheritance be replaced by composition?

Research Question 4 (RQ4): 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!!!!!!!!!!!!!!!

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!!!!!!!!!!!!!!!

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 Results

For Research Question 1: They conclude that late-bound self-reference plays a significant role in the systems we 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 (RQ4): 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 child parent 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: ¡child, parent¿. 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 child parent pair in an inheritance relationship has user defined attribute if both of child and parent are system types. In the example, pair ¡Q,P¿ has 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 child parent pair in an inheritance relationship in Java can have one of the three attributes: CC (Class Class) - both child and parent are classes, CI (Class Interface) - child is a class and parent is an interface or II (Interface Interface) - both child and parent 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 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 child type calls a method or accesses a field of its parent 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 child type itself, but it happens in another type, on an object of child type. According to original study, the class in which the external reuse occurs may not have any inheritance relationship with the child or parent 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 child type is supplied where an object of parent 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 in subtyping. 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 parent 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 parent type (parent-method) makes a call to another method (child-method) which is overridden by the child type. When an object of child type calls the parent-method, the child-method of the child type will be executed. This case is called *down*call, because a child type is found under the parent 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 child parent pairs which can not be placed under any other inheritance definition. In this case, we search for a sibling of the child which has a subtype relationship with the parent. If we can find such a sibling, we assume that the parent is used as a category class, and the child 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 child parent pair has constants attribute if the parent only contains constant fields (i.e., fields with `static final` attribute). The parent should either have no grandparents or if it has grandparents, the pair parent grandparent 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 child parent pair will have the framework attribute if it does not have one of the external reuse, internal reuse, subtype or downcall attributes and the parent is a descendant of a third party 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 child parent (for example : child type R, and parent type S) pairs which adhere to the following:

1. S is parent 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 child parent pair occurs when a parent has nothing declared in it. Moreover, just like the constants definition, the parent should either have no parents itself, or if it has parents, parent grandparent pairs should all have marker attribute. Parent should be defined as an interface and child may be a class or an interface.

```
public interface H { }

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

3.9.6 Super

A child parent pair will qualify for super attribute if a constructor of child type explicitly invokes a constructor of parent type via `super` call.

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

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 child parent pair should have explicit and user defined attributes.

`numExplicitCC`	Number of CC pairs.
`numCCUsed`	Number of CC pairs for which some subtype, internal reuse or external reuse was seen.
`perCCUsed`	`numCCUsed / numExplicitCC`.
`numCCDC`	Number of CC pairs for which downcall use was seen.
`perCCDC`	`numCCDC / numCCUsed`
`numCCSubtype`	Number of CC pairs for which subtype use was seen
`perCCSubtype`	`numCCSubtype / numCCUsed`
`numCCExreuseNoSubtype`	Number of CC pairs which do not have the subtype attribute, but which do have the external reuse attribute.
`perCCExreuseNoSubtype`	`numCCExreuseNoSubtype / numCCUsed`
`numCCUsedOnlyInRe`	Number of CC pairs which have neither the subtype nor the external reuse attribute, but which do have the internal reuse attribute.
`perCCUsedOnlyInRe`	`numCCUsedOnlyInRe / numCCUsed`
`numCCUnexplSuper`	Number of CC edges which do have super use and do not have any other types of inheritance usage.
`perCCUnexplSuper`	`numCCUnexplSuper / numExplicitCC`
`numCCUnexplCategory`	Number of CC edges which do have category use and do not have any other types of inheritance usage.
`perCCUnexplCategory`	`numCCUnexplCategory / numExplicitCC`
`numCCUnknown`	Number of CC edges which do have an inheritance relationship, but which do not have any of the inheritance attributes defined.
`perCCUnknown`	`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 child parent 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

5.1 Research Questions

5.2 Differences in the study set-up

5.3 Anlayisis Challenges

5.4 Results

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.

Chapter 9

Literature

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.