Inheritance Usage in Java: A Replication Study

C. Aytekin, T. van der Storm
cigdem.aytekin2@student.uva.nl, storm@cwi.nl

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 [TYN13]. 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 [TAD10]. 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.

1 Research Questions of the Original Study

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.)
RQ2:	To what extent is inheritance used in Java in order to express a subtype relationship that is necessary to design?
RQ3:	To what extent can inheritance be replaced by composition?
RQ4:	What other inheritance idioms are in common use in Java systems?

2 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

3 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.

For Research Question 4: They also report some 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 other usages do not occur frequently.

4 Definitions

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 is not defined in the system.

4.2 User Defined Attribute

The descendant ascendant pair in an inheritance relationship has user defined attribute if both of descendant and ascendant are system types.

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.

4.4 Explicit Attribute

The inheritance relationship is described directly in the code.

4.5 Internal Reuse

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

4.6 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.

4.7 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.

There are two interesting cases of subtyping usage in Java (sideways cast and this changing type). Please see the original article [TYN13] for the definitions for these two specific cases.

4.8 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.

4.9 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.

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.

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.

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.

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

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.

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.

5 Metrics

The metrics are explained elaborately in the website of the original study [TYN08]

6 Replication Study

6.1 Research Questions of the Replication Study

Our research questions directly refer to the four research questions of the original study 1 and can be summarized as: How do our results differ from those of the original study?

6.2 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.

Differences between the content of the byte code and the source code: It may 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.

Qualitas Corpus vs. Qualitas.class Corpus: The authors used the Qualitas Corpus [TAD10], we also use the Corpus, but the compiled version of it Qualitas.class Corpus [TMVB13b].

References

[TAD10] 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.

[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.

[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.