What Programmers do with Inheritance in Java

Ewan Tempero[1], Hong Yul Yang[1], and James Noble[2]

[1] University of Auckland, Auckland, New Zealand
e.tempero,hongyul@cs.auckland.ac.nz
[2] School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand kjx@ecs.vuw.ac.nz

Abstract. Inheritance is a distinguishing feature of object-oriented programming languages, but its application in practice remains poorly understood. Programmers employ inheritance for a number of different purposes: to provide subtyping, to reuse code, to allow subclasses to customise superclasses' behaviour, or just to categorise objects. We present an empirical study of 93 open-source Java software systems consisting over over 200,000 classes and interfaces, supplemented by longitudinal analyses of 43 versions of two systems. Our analysis finds inheritance is used for two main reasons: to support subtyping and to permit what we call external code reuse. This is the first empirical study to indicate what programmers do with inheritance.

1 Introduction

Inheritance is a concept that is given significant visibility to those learning object-oriented design, with texts on object-oriented programming often devoting several chapters to the subject (e.g. [4]). This raises the question of to what degree is inheritance actually used. In previous work we measured how much inheritance was used in a software system in terms of *how often* a developer *made the decision* to create an inheritance relationship between two types [27]. What we found was, on average, 3 out of 4 types were defined using some form of inheritance: inheritance is clearly important within Java programs.

Our results however do not tell the full story with regards inheritance use. While they tell us how much inheritance is used, they do not tell us what the designer uses it for, they do not tell us to what degree its use is *necessary*. It may be that some of the use we have observed is not appropriate use of inheritance. The main goal of the study presented in this paper is to determine whether or not this is the case. If the use is mainly appropriate, then this is important to know for two reasons. The first is that our earlier results become much more relevant in demonstrating the importance of inheritance. The second reason is the systems we analysed provide a benchmark for how inheritance is used.

Our previous study simply measured the amount of inheritance in programs, that is, what inheritance relationships exist between types. We only had to look at the `extends` and `implements` clauses of type declarations. In this study, we must look

at the implementation of each method to understand how those inheritance relationships were actually exercised. Whereas our last study considered the question "How do programs use inheritance?", in this study our question is **"What do programmers do with inheritance?"**, that is, having made the decision to use inheritance at the design level, what benefits follow from the use of inheritance. We are particularly interested in discovering unnecessary uses of inheritance, that is, cases where an inheritance relationship exists, but where it is not required. We address this question by studying a large corpus of open source Java systems.

This paper makes the following contributions:

- We develop a model of inheritance that represents how inheritance is used.
- We present the results of evaluating a corpus of open source Java systems against our model of inheritance, and make our data set available.
- Our overall conclusion is that inheritance in Java is mostly used to support subtyping and to permit external reuse. Additionally, a significant fraction of subclasses rely on polymorphic self-calls to customise their superclasses' methods' behaviour.

The rest of the paper is organised as follows. In the next section, we identify four research questions by considering previous discussions of why inheritance is used. Section 3 discuses inheritance in Java in detail, a necessary precursor to section 4, which presents our model of inheritance and the methodology of our study. In section 5 we present our results, and discuss their consequences in section 6. Finally, we present conclusions and discuss future work in section 7.

2 Motivation and Research Questions

There is much discussion in the research literature and trade press on inheritance as it applies to software, but there seems to be little on understanding how it is actually used. There seems to be a considerable uncertainty as to what it is or how to use it, if the number of web sites, blogs, and articles in the trade press are anything to go by. At the same time, there are quite public criticisms of inheritance, through writings with provocative titles such as "Why extends is evil" [15] or "Inheritance is evil, and must be destroyed" [23]. Examining such criticisms, we might conclude they are overstating the case based on a small set of examples, as there is little objective evidence that the problems they identify are widespread. Nevertheless, authors such as Gamma et al. instruct us to "Favor object composition over class inheritance" [12], which suggests they at least have seen enough questionable use of inheritance as to prompt such advice.

Within the research community, the recent focus has been on measuring inheritance with the hope of understanding the relationship between its use and some notion of quality of the software. For example Chidamber and Kemerer introduced the DIT and NOC metrics that measure two aspects of a individual class' use of inheritance [8, 7]. There have been several studies to establish the relationship between measurements from these and similar metrics and quality attributes such as maintenance ([18]) or prediction of fault prone classes [1, 3].

We detail three further related studies, those by Daly et al. [10], Cartwright [6], and Harrison et al [14]. Daly et al. examined the impact of depth of inheritance on

maintenance, with the conclusion that inheritance had a negative effect on maintenance time. In a rare replication Cartwright carried out a similar study, with results suggesting that inheritance had a positive effect on maintenance. Another replication was carried out by Harrison et al. Their results suggest that inheritance made it harder to modify systems, but that size and functionality of a system may affect understandability more than the "amount of inheritance" used.

There could be several explanations for these inconsistent results. For example, it could be that the systems under study were too small for inheritance to be the main factor affecting maintenance effort. It could also be possible that the uses of inheritance were not the same in all studies, or that depth of inheritance is not sufficient to characterise how inheritance is used. For example, as we have previously reported, different uses of overriding could explain the variation [26]. Another possibility, which we explore in this paper, is that programmers choose to use inheritance for different reasons. It could have been that the systems in the different studies used inheritance for different purposes, meaning the studies were not in fact comparing like with like.

In the early discussions of inheritance, there was much debate as to how it could be used, how languages should provide it, and whether it was even a good idea. Inheritance has been the subject of much discussion within the research community, The discussion explored such things as its interaction with encapsulation [22], how type systems are affected by inheritance [9], how it relates to other features such as genericity [19], or whether variants such as multiple inheritance are worth having [5, 29].

Of particular interest to us are two reports of how inheritance is or can be used. Meyer described what he regarded as 12 different valid uses of inheritance [20]. Taivalsaari discussed the many varieties and uses of inheritance, and provided a taxonomy for analysing inheritance [24]. Taivalsaari also observed that there seemed to be many benefits compared to other programming language features, but – crucially — described inheritance "*an incremental modification mechanism in the presence of late-bound self-reference*" and concluded that this seemed to be its most profound benefit. This conclusion is interesting. Late-bound self-reference — that method invocations on `this` (in Java) are also polymorphic as with any method invocation — is a feature of object-oriented languages that usually does not get much attention, and in some cases not much language support (e.g. Go [21]). This gives us our first research question:

RQ1: To what extent is late-bound self-reference relied on in the designs of Java systems?

Neither Taivalsaari nor Meyer provided empirical evidence to support their conclusions, and some of the uses of inheritance they described have no obvious operationalisation. These papers are also now quite old, raising the question as to how much of what they describe is still relevant, however they provide a useful starting point for understanding inheritance. In particular, Taivalsaari's taxonomy identifies three dimensions for analysing inheritance — what he called incremental modification, property inheritance, and interface inheritance. This provides a good basis for an empirical study, as we discuss in the next section.

Taivalsaari also observes that one view of inheritance is that it supports conceptual specialisation, but he and others have observed that most languages allow a class

that inherits to almost arbitrarily change its behaviour, and so in such cases the inheritance relationship would not reflect true conceptual specialisation. He comments, however, that "Subtyping, on the other hand, expresses conceptual specialization" and summarises the then thinking on the relationship between inheritance and subtyping. In Java, the subtype relationship is expressed using the Java inheritance mechanisms, and we will discuss this in more detail in the next section.

Taivalsaari also comments "In fact, the use of inheritance for conceptual specialization seems to be an ideal that is rarely realized" referring to Smalltalk and C++ libraries of the day. This is a surprising claim, as if true it would mean that the subtype relationship is "rarely" used. This has not been our experience with Java code, and in fact there is advice advocating using inheritance for subtyping. For example, Bloch says "Inheritance is appropriate only in circumstances where the subclass really is a *subtype* of the superclass" [2](p85). We know of no empirical evidence to support or refute such a claim. This leads to our next question:

RQ2: To what extent is inheritance used in Java in order to express a subtype relationship that is necessary to the design?

We will discuss the details of what it means to be "necessary" in section 4.

While Meyer (and to a lesser extent Taivalsaari) discuss a number of ways inheritance might be used, contemporary advice seems to be more conservative. As noted above, Gamma et al. caution against some forms of use. Bloch repeats the advice "Favor composition over inheritance" [2](Item 16) and provides a compelling example to support this. The advice is based on the argument that the form of inheritance referred to by Gamma et al. and Bloch fundamentally is an implementation decision. As such, inheritance breaks encapsulation, as observed previously by Snyder [22]. In fact Bloch shows a mechanical procedure to convert from this use of inheritance to composition (replacing inheritance with delegation).

Given that advice by such prominent authors is to avoid inheritance where possible, we might expect that there is infrequent use of form of inheritance they refer to. Specifically, we might expect that inheritance is avoided in favour of composition. It is difficult to tell when something is being avoided, but we can tell when it is not, so for our next research question we ask:

RQ3: To what extent can inheritance be replaced by composition?

There has been other discussion regarding use of inheritance either directly or indirectly. For example, Johnson and Foote discuss how features of object-oriented languages, including inheritance, can be used to develop reusable code [16]. In a similar vein, various specific uses have been recorded [12, 13]. We do not repeat this work, but are interested in identifying any inheritance idioms in common use:

RQ4: What other inheritance idioms are in common use in Java systems?

3 Understanding Inheritance

In order to measure how inheritance is used, we need to understand what it means. The study we present is of Java code, and so some of the details are Java specific. For example, by "inheritance" we mean when a Java type (class, interface, annotation, enum) `extends` or `implements` another type. While the details are Java specific, we believe the general concepts apply to most object-oriented languages.

```
class P {          class C extends P {
  void p() {         void c() {
    q();                 q();// internal
  }                    }
  void q() {       }
    ...
  }
}
class M {          class D extends P {
  void m(P aP) {     void q() {
    aP.p();              ...
  }                    }
}                  }

class N {
  void useReuse() {
    C aC = new C();
    aC.p();              // external
  }
  void useSubtype() {
    M anM = new M();
    C aC = new C();
    anM.m(aC);          // subtype
    D aD = new D();
    aD.p();              // downcall
  }
}
```

Fig. 1. Uses of Inheritance: "external reuse", "internal reuse", and "subtype". Modifiers have been elided.

3.1 Use of Inheritance

Inheritance is often presented as what Taivalsaari refers to as "property inheritance" — one class (the *child*) acquires properties of another (the *parent*) by inheriting them. This is one way in which inheritance supports **reuse**; the inheriting class can be written faster

because the inherited code does not have to be rewritten. This is illustrated in Figure 1. In the method `N#useReuse()` the method `p()` is invoked on an instance of `C`, however the code that is actually executed was not written for `C` but `C` has acquired it through inheriting (extending) `P`. Note that `p()` was accessed from outside the class `C`, which we refer to as **external reuse**. The method `C#c()` also makes use of an inherited method, but does so from within `C`, which we refer to as **internal reuse**. Figure 2 shows a more well-known example of internal reuse.

```
class Stack<E> extends Vector<E> {
  ...
  public E push(E item) {
    addElement(item);
    return item;
  }
  ...
}
```

Fig. 2. `Stack` demonstrates internal reuse by making a "self-call" (in bold) on its parent `Vector` as part of the implementation of one of its methods.

For external or internal reuse, every access to a member of a type is examined. If the member is not declared in that type, then it is some form of reuse. If the type that the member is declared in is an ancestor of type containing the code where the access takes place, then it is internal reuse, otherwise it is external reuse.

If a child class has all of the properties of a parent class, then it seems reasonable to expect that an object from the child can be used wherever an object from the parent is expected. This is Taivalsaari's interface inheritance dimension, although it is perhaps best known as the Liskov Substitution Principle [17]. This ability to substitute child objects for parent objects is formally recognised in the Java type system by regarding the type associated with the child class to be a **subtype** of that associated with the parent class. In the method `N#useSubtype()` in Figure 1, an instance of `C` is legally passed to the method `M#m(P)`, even though that method expects an instance of `P`. Without the subtype relationship, the code in `N#useSubtype()` would have to be duplicated in order to handle types other than `P`.

As noted in the previous section, many languages, Java included, allow the inheriting class to change what it inherits. The mechanism for doing so (for methods) is *overriding*. While doing so can result in inheritance no longer corresponding to conceptual specialisation, it can also allow quite sophisticated behaviour to be described. Taivalsaari includes this in his incremental modification dimension, but it is the late-bound self-reference aspect of it that is of interest to us. In figure 1, when `aD.p()` in `N#useSubtype()` executes, it invokes `P#p()` which in turn uses the (implied) self-reference to invoke `q()`. However in this case, the late binding of the self-reference

means that it is actually `D#q()` that is called. Late bound self-reference means one method can call another "below" it in the inheritance hierarchy, which we refer to as a **downcall** for brevity.

The uses described above come from the standard descriptions of inheritance. We are aware of other possibilities. One idiom is the **constants** interface, where an interface is a repository of useful constants, and any class "implementing" it can use those constants without the need to qualify them. While this practice is no longer recommended [2], it does represent a use of inheritance. Another idiom is the so-called "marker" interface. Such interfaces have no members, but it is necessary that classes implement to indicate they have certain capabilities that have no associated methods.

4 Methodology

Our goal is to determine why developers have made the decision to create an inheritance relationship between two types, that is, what *purpose* did they likely have in mind? We can infer this by examining the use they make of the relationship. We then measure the uses with respect to a collection of systems.

4.1 Modelling Inheritance

Metrics can be defined by different means [11]. One means is to base the definitions on a model of what is to be measured. The measurements we present in this work come from metrics based on the model of how inheritance is used in software, which we call the *inheritance graph*.

Measurement assigns numbers to attributes of entities, and it is the entities that we model. We want to measure the software that makes up what we generically refer to as a "system," however defining exactly what this is is difficult, as we have discussed elsewhere [25]. To resolve these difficulties we consider a system to be just those types that were created for that system. This excludes the Java Standard API and third-party libraries, a decision whose consequences we discuss further in Section 4.2.

For this study, we limit the types to just classes and interfaces, and furthermore, for classes we do not include exceptions (generally, types that are descendants of `java.lang.Throwable`). Enums, annotations, and exceptions are all defined using inheritance and so this use of inheritance is not a choice by the developer.

For a given system, its inheritance graph is a directed graph where the vertices include any type (class or interface) associated with the system implementation and the edges connecting these types that have some kind of inheritance relationship (`extends` or `implements`). This means we do not model edges between system types and non system types (third-party code). The vertices are named by the fully qualified name of the type they represent.

The edges have a set of attributes that capture the information for this study, which defined below. Direct metrics are then defined in terms of boolean expressions describing the presence or absence of attributes on edges. Some attributes represent properties inherent in the system code, some represent what we have observed in the system code

in terms of why the inheritance relationships are needed, and some represent information that has been established only by heuristics. We will indicate which applies when necessary. In the interests of brevity, we will uses phrases such as "subtype edges", by which we mean "inheritance relationships that we observed were relied on for the purpose of supporting the subtype relationship."

CC, CI, II: An edge will have one of these attributes if it represents a Class-Class (`extends`), a Class-Interface (`implements`), or an Interface-Interface (`extends`) relationship between system types.

External Reuse: An edge from types `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 declared in `T`.

The class `E` is using a member of `S` that was not declared in `S`, which is only possible because `S` has an inheritance relationship with `T`, so the inheritance relationship is necessary for this to be possible. This definition does not assume `S` and `T` are classes, but we only discuss external reuse with respect to classes in this paper.

Internal Reuse: An edge from classes `A` (child) to `B` (parent) has the internal reuse attribute when a method declared in `A` invokes a method `m()` or accesses a field `f` on an object constructed from `A` and `m()` or `f` is declared in `B`.

Without the stated inheritance relationship, it would not be possible to invoke `m()` or access `f` in this way.

Subtype: An edge from types `S` (child) to `T` (parent) has the subtype attribute when there is a class `E` (which could be `S` or `T`) in which an object of type `S` is supplied where an object of type `T` is expected. Within `E`, this might be assigning an object of type `S` to a variable declared to be type `T`, passing an actual parameter of type `S` to a formal parameter of type `T`, returning an object of type `S` when the formal return type is `T`, or casting an expression of type `S` to type `T`.

Without the stated inheritance relationship, `S` would not be a subtype of `T`, and so the substitution would not be possible. This means that this relationship is necessary for the correct behaviour of the code.

Downcall: An edge from classes `C` (child) to `D` (parent) has the downcall attribute when a method `c()` declared in `D` invokes a method `m()` that is declared in `C`.

The inheritance relationship in necessary for `c()` to invoke `m()`. The method `m()` must be declared in `D` or an ancestor of `D`, so `c()` is making a self-call to `m()`, but `C` overrides that declaration. The object on which the invocation takes place must be constructed from `C` or one of its descendants.

Framework: An edge from types `P` to `Q` that does *not* have external reuse, internal reuse, subtype, or downcall, has the framework attribute if `Q` is a descendant of a third-party type. (See also Section 4.3.)

Constants: An edge from types `E` to `F` has the constants attribute if `F` has only fields declared in it and the fields are constants (`static final`), and all outgoing edges from `F` either have the constants attribute or are to `java.lang.Object`.

The type `F` can be either an interface or a class.

Marker: An edge from type `G` to interface `H` has the marker attribute if `H` has nothing declared in it, and all outgoing edges from `H` have the marker attribute.

ant-1.8.1 antlr-3.2 aoi-2.8.1 argouml-0.30.2 aspectj-1.6.9 axion-1.0-M2 c_jdbc-2.0.2 castor-1.3.1 cayenne-3.0.1 checkstyle-5.1 cobertura-1.9.4.1 colt-1.2.0 columba-1.0 derby-10.6.1.0 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-20100806 freecol-0.9.4 freecs-1.3.20100406 galleon-2.3.0 ganttproject-2.0.9 heritrix-1.14.4 hibernate-3.6.0-beta4 hsqldb-2.0.0 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-2.4 jmoney-0.4.4 joggplayer-1.1.4s jparse-0.96 jpf-1.0.2 jrat-0.6 jre-1.5.0_22 jrefactory-2.9.19 jruby-1.5.2 jsXe-04_beta jspwiki-2.8.4 jtopen-7.1 jung-2.0.1 junit-4.8.2 log4j-1.2.16 lucene-2.4.1 marauroa-3.8.1 maven-3.0 megamek-0.35.18 mvnforum-1.2.2-ga myfaces_core-2.0.2 nakedobjects-4.0.0 nekohtml-1.9.14 openjms-0.7.7-beta-1 oscache-2.4.1 picocontainer-2.10.2 pmd-4.2.5 poi-3.6 pooka-3.0-080505 proguard-4.5.1 quickserver-1.4.7 quilt-0.6-a-5 roller-4.0.1 rssowl-2.0.5 sablecc-3.1 springframework-1.2.7 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-3.7.2 xalan-2.7.1 xerces-2.10.0

Fig. 3. Systems studied, including version identifier.

Super: An edge from class `K` to class `L` has the super attribute if a constructor for `K` explicitly invokes a constructor in `L` via `super`. (See also Section 4.2.)

Generic: An edge from type `R` to type `S` has the generic attribute if there has been a cast from `Object` to `S` and there is an edge from `R` to some (non-`Object`) type `T`. (See also Section 4.3.)

4.2 Study Details

We studied the 93 open-source Java systems from the 20101126 release of the Qualitas Corpus [25] listed in Figure 3. Not all systems from this release of the corpus were included as the tools we used had memory limitations that restricted the size of the systems that we could analyse. Table 1 gives provides statistics of some of these systems.

We also studied the history of two systems: `ant`, with 20 releases from version 1.1 to 1.8.1, and `freecol`, with 23 releases from version 0.3.0 to 0.9.4. We chose these two systems due to having the data for all releases in the corpus, and because they come from quite different domains (`ant` is a build tool with a plug-in architecture and managed through XML documents; `freecol` is a strategy game with a graphical interface, multi-media components, client-server architecture, and network communication).

The details of the systems can be found on the corpus website, in particular details of how we identified types belonging a given system. We analysed the bytecode of these systems. While most of what was needed for the analysis is in the bytecode, there is some loss of information as discussed below (Section 4.3).

There are several different kinds of analysis performed. To determine the subtype attribute, we first examine the code for where substitution can occur. The specific cases we detect are: passing a parameter, returning a value, assignment, and cast. For example, if the declared return type of a method is `T`, but the `return` statement references a variable of a different type `S`, then there must be a subtype relationship between `S` and

Table 1. Statistics for representative subset of systems studied (version elided). **Types** — number of types (including nested) in the system; **KLOC** — non-commented non-blank lines of code (thousands); **CC**, **CI**, **II** — number of the respective kinds of edges.

System	Types	KLOC	CC	CI	II
ant	1202	108	672	290	18
aspectj	3127	412	1142	626	110
derby	2755	593	697	525	91
drjava	5051	62	1269	1119	86
fitjava	85	2	43	0	0
freecol	1542	82	392	127	3
jrat	255	14	34	37	0
jre	11736	831	5735	5102	799
jruby	5783	160	3671	735	12
jsXe	144	9	11	3	0
jtopen	3482	397	1347	687	16
megamek	2969	259	1283	213	10
mvnforum	4194	51	18	45	0
nakedobjects	4963	110	1307	732	349
nekohtml	2016	7	10	8	0
trove	715	2	126	269	0
weka	2125	224	516	1047	10

```
class MapboardAction extends FreeColAction {
  ...
}
class LoadAction extends MapboardAction {
  ...
}
```

Fig. 4. Classes from `freecol` illustrating indirect and implicit edges. (Package name and modifiers elided)

`T`, and furthermore the relationship is necessary for the code to compile. An example of the parameter passing case is shown in method `N#useSubtypes` of Figure 1, where an object of type `C` is passed to a method whose formal parameter type is `P`.

The subtype analysis uses what is essentially a reachability analysis for all reference type definitions to all uses. Where the type of the def does not match the type of the use, and since we know the code compiled (as we are analysing bytecode), we know we are dealing with subtype use. The tool we use is based on the Soot framework [28].

Having identified when subtype substitution is used, we then match the relationships required to the relationships expressed in the code. This is necessary because those required may not be explicit. For example, given the declarations from `freecol` shown in Figure 4, it is possible to substitute `LoadAction` for `FreeColAction` despite the fact that there is no direct subtype relationship between these two types. Any code that depends on such a substitution being possible will result in the edges representing the

```
C implements P1
C implements P2
void method(P1 p1) {
  P2 p2 = (P2)p1; // sideways cast
}
```

Fig. 5. Example of a "sideways" cast.

```
class P {
  // A constructor expects type P
  private A anA = new A(this);
}
class C extends P {
  // C is passed to A constructor
}
```

Fig. 6. Example showing `this` changing type.

`LoadAction` – `MapboardAction` and `MapboardAction` – `FreeColAction` relationships being given the subtype attribute.

There are also two special cases that must be addressed. One is the "sideways" cast, where Java allows what looks like a cast between unrelated types. In the example in Figure 5, the cast will be successful provided an instance of `C` is passed to the method. Such situations represent use of the subtype relationship between `C` and its parents, and so must be detected in order to correctly identify all subtype uses.

The other case involves the pseudo variable `this`, which can change its type in the presence of inheritance. This change can indicate the use of a subtype relationship. In the example in Figure 6, the use of `this` in the constructor call to `A` indicates the use of the subtype relationship between `P` and `C`.

For external or internal reuse, every access to a member of a type is examined. If the member is not declared in that type, then it is some form of reuse. If the type that the member is declared in is an ancestor of the type containing the code where the access takes place, then it is internal reuse, otherwise it is external reuse.

If a method invocation is a self-call (whether on a method declared in that class or due to internal reuse), then it could be a downcall. In such cases, all descendants are examined and if the method being invoked is overridden, then we make the conservative assumption that a downcall can take place.

When doing the analysis, we ignore the use of default constructors. There must always be a call by a class (in its constructors) to the constructor in its parent. This would look like internal reuse, however it would often happen without any intervention by the programmer. As we want to understand what decisions programmers make with respect to inheritance, we separate out these calls. Since calls via `super` are somewhat under

```
package org.jgraph.graph;
...
import javax.swing.TransferHandler;
...
public class GraphTransferHandler
    extends TransferHandler {
  ...
}
```

Fig. 7. The "framework" problem. Uses of some inheritance relationships may not be visible in the analysed code.

the control of programmers, we distinguish those from calls to the default constructor with the super attribute.

Note that we take a very liberal view of when reuse occurs. Whether external reuse or internal reuse, we will mark inheritance relationships (edges) as such if only one member is used. This means that a child class might inherit 100 methods from its parent, and only 1 of those 100 methods might be used, but we would still consider this an indication of reuse.

4.3 Analysis Challenges

Because we do not consider the use of third-party libraries or the Java Standard API, the purpose of some inheritance relationships cannot be determined. For example, consider Figure 7 showing the `jgraph` class `GraphTransferHandler`. This class inherits from a class in the Swing framework and so it being substituted for `TransferHandler` may only be visible in the Swing implementation, but not in the code we analyse. This also means that descendants of `GraphTransferHandler` might also be substituted for `TransferHandler`, but our analysis would not detect this. When we suspect this situation is possible, the edge will get the "framework" attribute.

Another limitation of our analysis is when generic types are implemented using casting to and from `java.lang.Object`. An object of one type can be put into a generic container and then cast to a different type on its removal. This behaviour depends on a subtype relationship existing between the two types. When we suspect this situation is possible, the edge will get the generic attribute.

Bytecode does not always directly map on to source code, and this can effect some of our analysis. One example is shown in Figure 8. Method invocation is indicated in the bytecode with an instruction such as `INVOKEVIRTUAL`, and will include information on which method is being invoked. In the case of the example, when `U` is compiled, we might expect that it is `C#parent()` that is written into the bytecode as the method being invoked, as that is a correct representation of what should happen. However, some compilers will write `P#parent` into the bytecode. As this information is not used in method lookup, the difference does not cause problems in execution, but it does affect model fidelity. It will mean it is possible for some relationships for which *external reuse* use occurs in the source code to to not be attributed as such in the model. We have been

unable to reproduce this case ourselves with compilers we have access to, but do know it happens, albeit rarely, in code in the corpus.

```
class P {
  public void parent() { ... }
}
class C extends P {
  public void child() { ... }
}
class U {
  void user() {
    C aC = new C();
    aC.parent();
  }
}
```

Fig. 8. In some cases, the indicated invocation will show `P` as the invoked type, not `C`.

5 Results

We present our results here organised around our research questions. Due to the volume involved, we cannot present all the data[3], so we present those results that best indicate trends, and of systems of interest such as `ant`, `freecol`, and `jre` (the largest system studied by any measure of size).

5.1 RQ1: Late-bound self-reference

Downcall edges must be CC edges. Figure 9 shows, for each system, the proportion of CC edges that are downcall edges. The systems are in increasing order of the number of CC edges. The order was chosen to determine whether or not there was a trend with respect to this size metric (and the equivalent order will be used for most other charts). The x-axis labels indicate the order of magnitude of the number of edges (e.g. "00" indicates from 10^2 up to (not including) 10^3 edges).

The results indicate quite wide variation between systems, from zero (`freecs` — 61 edges, `jasml` — 21, `megamek` — 1283) to to 86% (`jFin_DateMath` — 22). The median is 34% (`aoi`). For the systems of interest, `ant` had 28%, `freecol` had 35%, and `jre` had 9%.

[3] Available from `http://www.cs.auckland.ac.nz/~ewan/qualitas/studies/inheritance`.

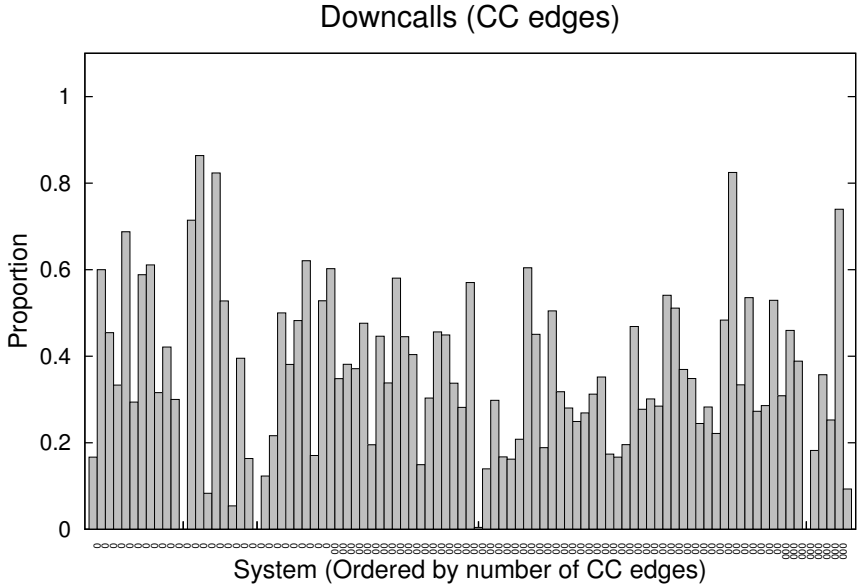

Fig. 9. Proportion of CC edges over which downcalls may occur. X-axis labels indicate order of magnitude of number of CC edges.

Figure 10 shows the downcall proportions for the releases of `ant` and `freecol`. Both show not-trivial uses of downcall (over 20% for `ant`, and nearly 40% for `freecol`), but both also show quite large changes between some releases (increasing and decreasing).

There is no obvious trend with respect to size as measured by number of CC edges between user-defined types. Our conclusion is that late-bound self-reference plays a significant role in the systems we studied — around a third (median 34%) of CC edges involve downcalls.

5.2 RQ2: Subtype relationship

CC, CI, and II edges can all be subtype edges. We present the results for each kind of edge separately. For the CC edges, we first identified those CC edges that were at least one of subtype, external reuse, or internal reuse. Figure 11 shows (bottom segment, "ST") the proportion of those CC edges that are subtype edges (other values will be discussed below), with the systems ordered as in Figure 9.

Again we see wide variation, with the smallest being at 11% (`checkstyle`), two with 100% (`magemek` with 1283 CC edges, `jasml` with 21), and the median at 76% (`jmeter`). The largest system (`jre`) had a measurement of 95%, indicating that almost all of the `extends` relationships between classes had some subtype use within its implementation.

Figure 12 shows the subtype use of all CI edges in a system. The bottom segment ("ST") indicates proportion of CI edges for which subtype use was seen. The second

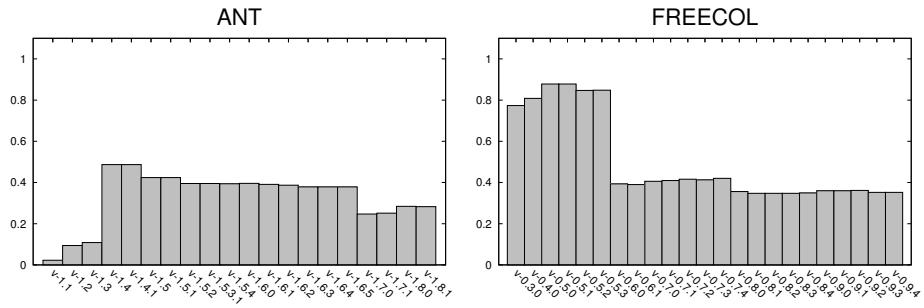

Fig. 10. Downcalls `ant` and `freecol` (x-axis is release order).)

segment ("SUS") indicate the proportion of edges for which we suspect there is subtype use, but limitations of our analysis means we did not directly observe such. We include these so as to not bias our results against use of subtype. The other segments will be discussed below.

There is one system (`fitjava`) with no CI edges. Of the remainder, there are 3 systems (`nekohtml`, `jsXe`, and `joggplayer`) for which all CI edges were subtype edges, however they all had fewer than 10 CI edges. In a further 4 systems (`jasml`, `jmoney`, `jparse`, `javacc`) all CI edges where either subtype, suspected of being subtype edges. The median use was 69% (`checkstyle` considering subtype only, or 85% (`megamek` if including suspected subtype edges. For the other systems of interest, `jre` had 82% being subtype edges, or 91% including those suspected, `ant` had 63% and 78% respectively, with `freecol` having 83% and 94%.

Figure 14 shows the use of II edges, with the systems ordered in increasing order of number of II edges. There are 23 systems with no II edges (and so have no values in the chart), and 51 systems in total with fewer than 10 edges. As before, the bottom ("ST") segment shows the subtype edges. Of these, 13 systems had all II edges being subtype edges, however the largest (`jhotdraw`) had only 14 edges. The median use was 63% (`findbugs`). Of the systems of interest, `jre` had 71% (of 799 II edges), `ant` had 94% (18), and `freecol` had 67% (3). The system with the second largest number if II edges (`nakedobjects` with 349) had 66%.

Our conclusion is that at least two thirds of all inheritance edges are used as subtypes in the program — inheritance for subtyping is not as rare as Taivalsaari implies [24].

5.3 RQ3: Inheritance vs. Composition

Our interest here is identifying use of inheritance that could have been changed to composition using a procedure such as that proposed by Bloch [2]. This procedure would not apply when the purpose for using inheritance was to establish a subtype relationship, so we need to identify those edges that are internal or external reuse, but not subtype.

The procedure described by Bloch can be tedious to apply for parents with many members, and so it could be argued it is not always practical, however it is much simpler (and hence requires less effort) when only internal reuse is involved. Consequently we identify also those edges that are internal reuse only.

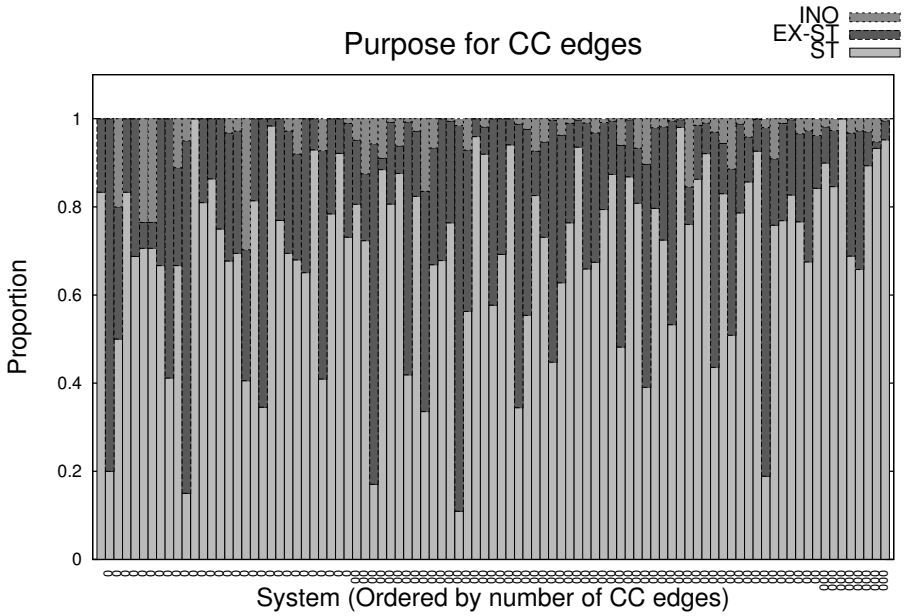

Fig. 11. CC edges that are subtype edges (ST), external reuse edges but not subtype edges (EX-ST), or only internal reuse edges (INO).

Figure 11 shows, as well as the proportion of subtype edges, those edges that are external reuse (and possibly internal reuse) but not subtype, and those edges that are internal reuse only.

The system with the largest proportion of external reuse (but not subtype) edges was `checkstyle`, with 88% of 193 CC edges. The median was 22% (`javacc`, 88 edges). For internal reuse edges only, the largest was 30% (`jpf`, 37 edges) and the median was 2% (`lucene`, 446 edges). There were 24 systems with no internal reuse only edges, the largest being `megamek`. For other systems of interest, `ant` had 20% external reuse and 1% internal reuse, `freecol` had 13% external and less than 0.5% internal, and `jre` had 4% external and less than 0.5% internal.

Figure 15 shows the same data as Figure 11 for `ant` and `freecol`. One point to keep in mind when interpreting these charts is that `ant` grew from 44 CC edges to 672 edges, and `freecol` grew from 53 to 392 edges. The proportions are remarkably constant after about the 5th release (`ant-1.5` has 401 CC edges, `freecol-0.6.0` has 239). It would be interesting to know why this is so, for example is it due to its architecture, or some other reason.

Our conclusion is that there is generally opportunity for replacing inheritance with composition, with 22% or more uses of inheritance between classes needed for external reuse but not subtyping in half the systems we examined. For internal reuse edges only, there are many fewer opportunities for replacing inheritance with composition, but they do exist for 2% or more of such uses in half the systems. We cannot say whether replacing inheritance with composition is worth the effort because we have no way to quantify the costs of not doing so. We do believe, however, that the prevalence of this

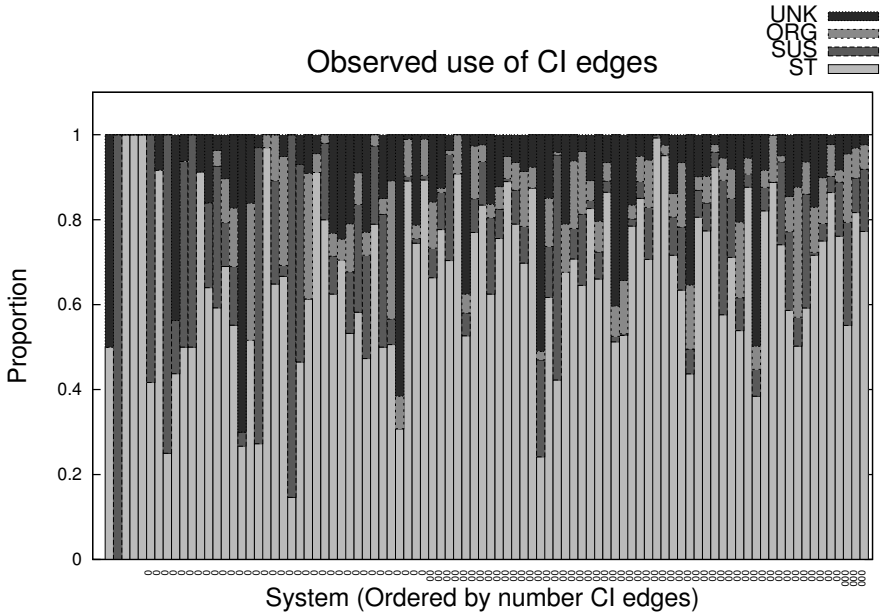

Fig. 12. CI edges that are subtype edges (ST), suspected to be subtype (SUS), organisational (ORG), or of unknown purpose (UNK).

use of inheritance is high enough to justify further research effort needed to understand how to quantify the costs, and also to give greater emphasis in teaching to avoid such uses of inheritance.

5.4 RQ4: Other uses of Inheritance

It is instructive to consider those inheritance relationships that do *not* support at least one of external reuse, internal reuse, or subtype, as this helps us understand to what other purpose developers might use inheritance. For the remainder of this section we will only be referring to these as yet uncategorised edges.

We noted the possible use of interfaces (or classes) solely to define *constants*, and the use of marker interfaces. For the former, only 13 systems had CC edges where the parents held constants and 5 of these had more than 1%, the largest of these being `fitlibraryforfitnesse`, with 13% of 259 edges. For CI edges 45 systems had no occurrences, and 18 had more than 10%. The largest was 100% by `jasml` (2 edges), but 4 systems had more than 50%. While some of these results were clearly due to such things as parser-generators (or similar), they do indicate that this idiom is fairly common, remembering that subtype, external reuse, or internal reuse edges can also support this idiom.

For use of *marker* interfaces, 61 systems had no occurrences of CI edges being relationships to marker interfaces, however of those that did, they predominantly came from the larger systems. The largest proportion was 47% (`jext` with 43 CI edges), but (for example) `weka` had 14% of 1047 edges in this category. These results suggest that the use of marker interfaces is also a common (sole) reason for using inheritance.

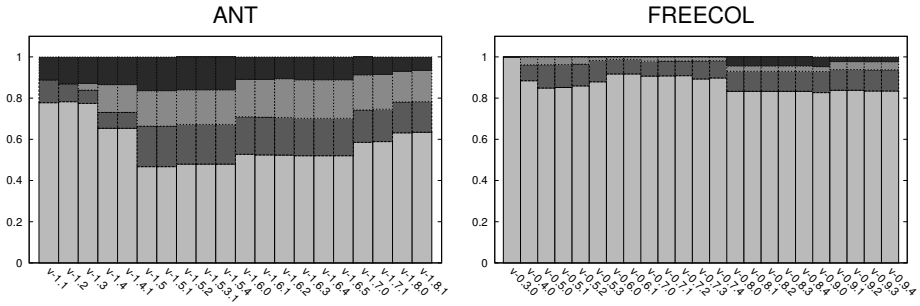

Fig. 13. Use of CI edges for `ant` and `freecol` (legend as for Figure 12, x-axis is release order).

As discussed in Section 4.3, some edges may be *framework* or *generic* edges. For CC edges, 58 systems had no occurrences of framework or generic edges that were also not constants or marker edges. Of those that did, 16 had less than 1%. The largest was 17% (`webmail`, with 24 CC edges). For CI edges, 38 had none, 8 had more than 10%, with `oscache` having 58% (of 12 edges). For II edges, only `jmeter` had any (5% of 20).

Of the remaining CC edges, one pattern we noticed was children whose only use of the relationship with the parent class was via calling a non-default constructor (via `super`). Another pattern we noticed for all edge types was one edge may appear to have no purpose, but an edge from a sibling to the common parent was one of subtype, internal reuse, or external reuse. It could be that the parent was playing an organisation role, indicating types that are conceptually related but which relationship may play no role within the implementation. We report such edges as *organisational* ("ORG" in figures 12, 13, and 14).

Figure 16 shows the unused CC edges with (light gray) showing those whose sole use is super constructors, (medium gray) showing remaining edges that were organisational, and edges with no purpose was identified in our analysis in (dark). As can be seen, the measurements are mostly small. The large value (38%) for calls to super constructors is in `trove`, and are by classes that appear to be all generated. There are 57 systems where we could associate some purpose to all edges, with 15 having more than 1% and being mostly systems with a large number of CC edges. The largest is `jre` with 8% of CC edges having no obvious purpose.

Figure 12 also shows our analysis of those CI edges that are not subtype edges. The third segment ("ORG") show those CI edges that are organisational, and the top segment ("UNK") show the proportion of CI edges for which we could find no purpose. Only 9 systems had no edges for which we could find no purpose (`jparse` was the largest of these having 41 CI edges). The system with the most such edges was `jre` (470 edges, 9% of its CI edges), with the median number of edges being 20 (`velocity`, 32%). The system with the largest proportion of such edges was `c_jdbc` (70% of 30 edges), and the median proportion was 15% (of the 113 CI edges of `pooka`).

Figure 14 also shows the other uses we observed of II edges. The segments indicate proportions for the different categories as in Figure 12, with the addition of a (second) segment ("RE-ST") are "reuse" edges that are not subtype. In the context of II edges

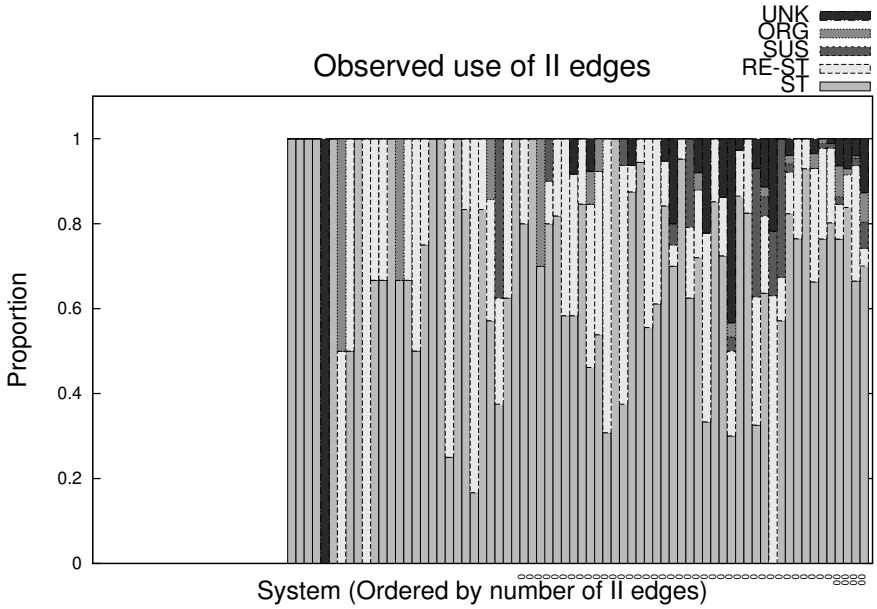

Fig. 14. II edges that are subtype edges (ST), reuse (RE-ST), suspected subtype (SUS), organisational (ORG), or unknown purpose (UNK). Systems with no II edges have no values (at left end).

this means that a method was seen to be invoked on an interface type, but in fact that edge was declared in an ancestor interface. While the majority are subtype, the reuse category stands out. This category shows interfaces whose sole purpose is to indicate where there is shared behaviour between types in the implementation. There are 54 systems with such edges. The maximum value was 100% (`colt`, 3 edges). Of the 70 systems with any II edges, the median value was 17% (`jgrapht`, 6 edges). The largest system `jre` had 4% (of 799 edges), `ant` had 6% (18 edges) and `nakedobjects` had 27% of 349 edges.

For edges with no purpose, this was true of all 2 of `checkstyle`'s II edges and 13% of `jre`'s edges. Only 21 systems had any such edges.

Our conclusion is that our conservative inheritance model classified over 58536/67529 (87%) of all edges in our graph (38122/39973 or 95% of all CC relationships) — as either subtype edges, external reuse, and internal reuse, that is, subtype, external, and internal reuse explain most of the inheritance relationships in our corpus.

6 Discussion

As indicated, there are some limitations to our analysis, so the natural question is to what degree do they threaten the validity of our conclusions. The first point to make is that the results for RQ2, and RQ3 indicate edges for which we actually observed subtyping and reuse respectively. That is, we may have false negatives, but no false positives, so the analysis is conservative with respect to these questions.

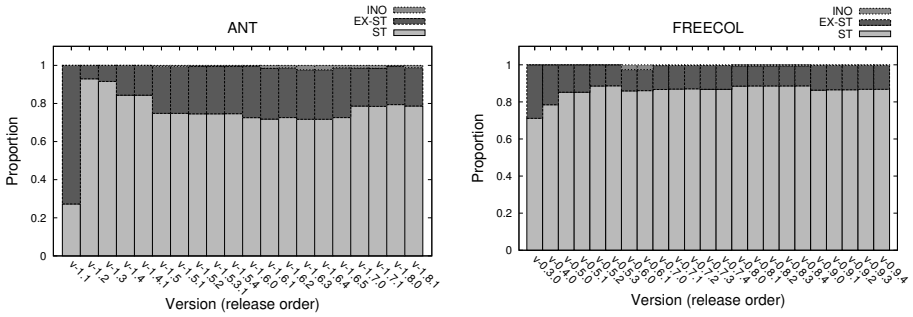

Fig. 15. CC edges that are subtype edges, external reuse edges but not subtype edges (EX-ST), or only internal reuse edges (INO) over time for `ant` and `freecol`.

Regarding RQ1, it is possible our results overstate reality, since we assume that if a self-call exists, and a descendant overrides that method, that a downcall will occur at the self-call. Since this depends on the run-time behaviour of the system, we cannot be sure that this will always be the case. We did limited manual inspection and found no overestimation.

For RQ4, we are quite confident about the accuracy of the measurements for classes and interfaces containing only constants, and for use of marker interfaces. For those edges that we reported for framework and generic use, if our assumption is correct and they are in fact used within frameworks or for generic types, then this would indicate the degree to which our subtype results are under-reporting the true situation. We know this is the case for a number of systems through manual inspection.

The most uncertainty exists for those edges we cannot easily classify, those that we report as UNK. The CI category of edges had the highest incidence of such edges (figure 12), which is surprising, given there is no obvious purpose to a class implementing an interface other than use the subtype relationship. Only 9 systems had no edges in this category (`jparse` was the largest of these having 41 CI edges). The system with the most such edges was `jre` (470 edges, 9% of its CI edges), with the median number of edges being 20 (`velocity`, 32%). The system with the largest proportion of such edges was `c_jdbc` (70% of 30 edges), and the median proportion was 15% (`pooka`, 113 CI).

We have manually examined a number of unused CI edges. We classified them as one of: implementations provided by the system for use by other clients; intended for future development; unable to classify; and having no good reason for existing.

For the first category we noted that the systems were frameworks or libraries, there were no names declared of the relevant types, the names had “Default” or “Adaptor” in them (e.g. `DefaultActionNameBuilder` in `struts`), or we had some other reason (e.g. comments) to think the types were intended for clients of the framework, not the framework itself. The `jre` is a good example of this. Given its purpose, it should be unsurprising that some of the types it provides are not used within it, and so if those types are defined using inheritance we cannot expect to see the use of the inheritance relationship in any way within `jre`. In fact, we observed subtype use of 95% of `extends` edges in `jre`.

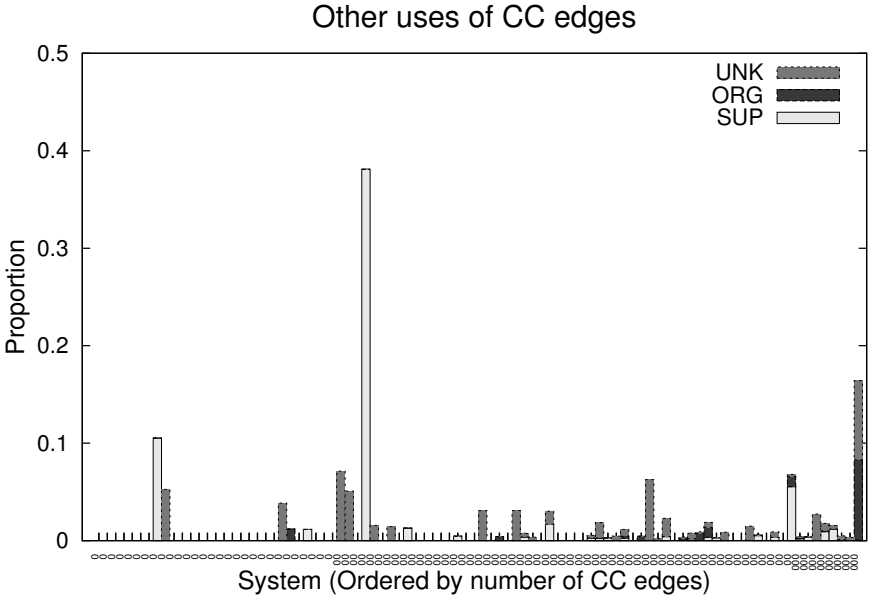

Fig. 16. Uses of CC edges that are not subtype, external reuse, or internal reuse edges — super constructor (SUP), organisational (ORG), or unknown purpose (UNK).

This represents the other side of the "framework" issue we discussed in section 4. Because we do not include third-party libraries in our analysis, we cannot detect uses of inheritance that cross the boundary between system code and third-party code.

The last three categories were ever more subjective. The category "future development" was adjudged if we had some reason (e.g. comments) to believe the relationships would be used in future releases. The category "unable to classify" we choose if we had some reason to not be able to make a judgement. For example, the possibility of reflection meant it was possible we missed some cases (although many would in fact be correctly identified by our tools), or the complexity of the design and the limited time available meant we could not come to any conclusion. The last category (no good reason) we chose when we could find no reason for the relationship (e.g. when we found no declarations of the interface type but many declarations with the implementation types). While this discussion refers specifically to CI edges, we performed a similar analysis for a subtype of CC and II edges as well.

Despite the subjectivity of some of our analysis, we did see cases where even with the most generous interpretation there seemed no reason for having an inheritance relationship. Nevertheless, such occurrences were fairly rare, and so we feel we can quite confidently state that there is little evidence of systematic unnecessary inheritance.

There is quite wide variation in the size of the systems. Using the total of CC, CI, and II edges as a size metric (see also Table 1), `jre` was the largest (11636), followed by `jruby` (4418), and `drjava` (2474). Most systems were only 10% of `jre` (75 with fewer than 1000 edges, the smallest `jsXe` had 14). Despite this, no one system

dominates any of our measurements, suggesting that how inheritance is used is not determined by the size of the system.

The longitudinal studies (figures 10, 13, and 15) show some abrupt changes. They all correspond to significant changes in the number of inheritance relationships (CC, CI, and CC respectively), and generally to significant changes in the overall code bases. This all points to changes in the design, however our measurements cannot show what led to those changes. That will require much deeper analysis, both of the code base and of the developers' thinking.

We have only considered relationships between system types, and so framework relationships (e.g. Figure 7) are not modelled. Due to the framework issue, we believed we would not be able to adequately represent the situation, and indeed when we include such edges we see generally a lower proportion of subtype edges. Nevertheless, they do indicate decisions made by the developer and we would like to be able to study these edges in more detail in the future.

We take a very liberal view of when we classified a relationship as for subtype or reuse. For example, there may be only one point in the implementation where subtype is needed, but many uses of external reuse, however we would show it as use of subtype. Our view is coarse-grained, but it does give an overall indication of how inheritance is used. With these results, we can now identify more specific questions of how inheritance is used, and they also help us restrict the systems we need to investigate to answer the questions. In particular, it would be valuable to revisit the studies done previously ([10, 6, 14]) but using (for example) degree of use of subtype versus external reuse as the independent variable, rather than DIT. Such studies would determine the validity of advice regarding composition versus inheritance.

While authors such as Meyer [20] and Taivalsaari [24] suggest there are many uses for inheritance, our studies suggests there are only two main uses in Java code. This could be due to how we classify relationships. More study is needed in this regard.

Our results also appear to disagree with Taivalsaari's observations. He characterised use of subtype as rare. It would be interesting to know whether this is due to the languages he referred to (Smalltalk and C++), due to the reasons programmers use inheritance having changed over time, or due to his observations being based only on his experience. Only further objective empirical studies will definitively answer such questions.

7 Conclusions

Our overall goal is to understand how design decisions impact the quality (however it is defined) of software. We are currently examining how inheritance is used by developers. In this paper we have presented a study of 93 open-source Java systems. We found about one third of subclasses rely on late bound self-reference (downcalls) to customise their superclasses' behaviour (RQ1). Java developers mostly use inheritance for subtyping, with about two thirds of inheritance relationships needed for this (RQ2). While there is not huge opportunity to replace inheritance with composition (RQ3), the opportunity is significant (median of 2% of uses are only internal reuse, and a further 22% are only

external or internal reuse). While there are other uses of inheritance, their use is not generally significant (RQ4).

Our results suggest there is no need for concern regarding abuse of inheritance (at least in open-source Java software), but they do highlight the question regarding use of composition versus inheritance. If there are significant costs associated with using inheritance when composition could be used, then our results suggest there is some cause for concern. We believe understanding these costs is an important open question.

This research also provides support for our previous work [27]. Our conclusion in that work was that there was a considerable amount of use of inheritance. We can now say that most of that use is justified. Whether this use was the best design choice remains to be seen, but it emphases the important of inheritance, at least for Java programmers. It also means that future research on the use of inheritance can use the same systems we have used without concern that unnecessary use of inheritance might taint the results.

There are many other possible avenues of research following from the work presented here. One of particular interest to us is to understand the rationale behind a programmer's use of inheritance. We will use qualitative techniques based on grounded theory to gain this understanding. It is also important that our research be replicated, including with other Java (particularly closed-source) systems, and systems in other languages.

References

1. Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.
2. Joshua Bloch. *Effective Java*. Addison-Wesley, 2nd edition, 2008.
3. Lionel C. Briand, John Daly, Victor Porter, and Jürgen K. Wüst. A comprehensive empirical validation of design measures for object-oriented systems. In *METRICS '98: Proceedings of the 5th International Symposium on Software Metrics*, pages 246–257, Washington, DC, USA, 1998. IEEE Computer Society.
4. Timothy Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley, 3rd edition, 2002.
5. Tom A. Cargill. The case against multiple inheritance in C++. *USENIX Computing Systems*, 4(1):69–82, Winter 1991.
6. Michelle Cartwright. An empirical view of inheritance. *Information and Software Technology*, 40:795–799, 1998.
7. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.
8. Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *OOPSLA*, pages 197–211, 1991.
9. William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '90, pages 125–135, New York, NY, USA, 1990. ACM.
10. 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, January 1996.
11. Norman E. Fenton and Shari L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998.

12. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley Publishing Company, One Jacob Way, Reading, Massachusetts 01867, 1994.
13. Joseph (Yossi) Gil and Itay Maman. Micro patterns in Java code. In *OOPSLA*, pages 97–116, 2005.
14. R. Harrison, S. Counsell, and R. Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *Journal of Systems and Software*, 52:173–179, 2000.
15. Allen Holub. Why extends is evil: Improve your code by replacing concrete base classes with interfaces. JavaWorld.com, August 2003.
16. Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June/July 1988.
17. Barbara Liskov. Keynote address — data abstraction and hierarchy. *SIGPLAN Notices*, 23(5):17–34, January 1987.
18. Dennis Manel and William Havanas. A study of the impact of C++ on software maintenance. In *International Conference on Software Maintenance*, pages 63–69, 1990.
19. Bertrand Meyer. Genericity versus inheritance. In *OOPSLA*, pages 391–405, New York, NY, USA, 1986. ACM.
20. Bertrand Meyer. The many faces of inheritance: a taxonomy of taxonomy. *IEEE Computer*, 29(5):105–108, May 1996.
21. Frank Schmager, Nicholas Cameron, and James Noble. GoHotDraw: evaluating the go programming language with design patterns. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 10:1–10:6, New York, NY, USA, 2010. ACM.
22. Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA*, pages 38–45, 1986.
23. Bernie Sumption. Inheritance is evil, and must be destroyed. Last accessed December 2012. `http://berniesumption.com/software/inheritance-is-evil-and-must-be-destroyed`, 2007.
24. Antero Taivalsaari. On the notion of inheritance. *Comp. Surv.*, 28(3):438–479, 1996.
25. 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. `www.qualitascorpus.com`.
26. Ewan Tempero, Steve Counsell, and James Noble. An empirical study of overriding in open source Java. In *Thirty-Third Australasian Computer Science Conference (ACSC2010)*, pages 3–12, Brisbane, Australia, January 2010. Volume 102 in the Conferences in Research and Practice in Information Technology (CRPIT) Series.
27. Ewan Tempero, James Noble, and Hayden Melton. How do Java programs use inheritance? An empirical study of inheritance in Java software. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 667–691, Paphos, Cyprus, July 2008. Springer Berlin / Heidelberg.
28. Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *CASCON*, pages 125–135, 1999.
29. Jim Waldo. Controversy: The case for multiple inheritance in C++. *USENIX Computing Systems*, 4(2):157–172, Spring 1991.