An Empirical Study of Overriding
in Open Source Java

Ewan Tempero  Steve Counsell  James Noble
Department of Computer Science  Department of Information Systems and Computing  School of Engineering and Computer Science
University of Auckland  Brunel University  Victoria University of Wellington
Auckland, New Zealand  Uxbridge, United Kingdom  Wellington, New Zealand
e.tempero@cs.auckland.ac.nz  steve.counsell@brunel.ac.uk  kjx@tecs.vuw.ac.nz

Abstract

Inheritance is a key feature of object-oriented programming. Overriding is one of the most important parts of inheritance, allowing a subclass to replace methods implemented in its superclass. Unfortunately, the way programmers use overriding in practice is not well understood.

We present the first large-scale empirical study of overriding. We describe a suite of metrics that measure overriding and present a corpus analysis that uses those metrics to analyse 100 open-source applications, containing over 100,000 separate classes and interfaces. We found substantial overriding: most subclasses override at least one method and many classes that only declare overriding methods. We also found questionable uses of overriding, such as removing superclass method implementations by overriding them with empty method bodies.

Keywords: Overriding, Inheritance, Object-oriented design

1 Introduction

Inheritance is one of the defining features of object-orientation and there are many learned articles, books, blogs, courses, and commentaries explaining what it is, and implicitly or explicitly advocating its use. Inheritance is also a source of confusion and unhappiness however, leading Holub to write an article with the title “Why extends is evil” (Holub 2003) and the Gang of Four to exhort us to “Favor object composition over class inheritance” (Gamma et al. 1994). Certainly inheritance is a multifaceted tool: supporting code reuse via subclassing; polymorphism via subtyping; domain modelling via concept hierarchies; and allowing methods to have multiple definitions via overriding. Software developers would like to know when and how inheritance should be used.

We recently conducted a study into the use of inheritance for subclassing and subtyping in Java programs (Tempero et al. 2008). This study found quite high uses of various forms of inheritance: three out of four types were defined using inheritance (not involving Object). That study considered only inheritance between types: it did not examine how those relationships were used in detail, and in particular did not consider how inheritance between classes affects the methods those classes declare.

While some of the criticisms regarding inheritance might be in fact regarding the use of overriding, this is not to say that overriding should be avoided altogether, and indeed one of the positive examples of use of inheritance is due to the fact that overriding is possible. This means that we need to establish how overriding is actually used, and identify those uses that are questionable or risky. In order to do this, we need some means to characterise how overriding is used. This paper represents a first attempt to answer these questions. Specifically, its contributions are:

1. Present metrics that indicate how overriding is used in a system.
2. Describe, in terms of measurements from the metrics, how overriding is used in 100 open-source Java systems, providing a benchmark for future discussions on the use of overriding.
3. Identify any questionable uses of overriding.

The rest of the paper is organised as follows. In Section 2 we present our motivation in more detail and discuss related work. Section 3 presents details of what metrics we used in our study and what we measured. The results of our study are presented in Section 4, and we give our interpretation of these results in Section 5. Finally we give our conclusions and discuss future work in Section 6.

2 Background

2.1 Motivation

The “anti-inheritance” arguments refer to so-called “implementation inheritance”, that is, when one class (in Java) extends another class and so allows the possibility of some of the implementation of that class to actually be provided by one of its ancestors. When examining the arguments against use of inheritance, the ability for the inheriting class to override what it inherits is clearly an important source of concern.

Gamma, commenting on “Favor object composition over class inheritance”, says “But we know that [inheritance is] brittle, because the subclass can easily make assumptions about the context in which a method it overrides is getting called” (Venners 2005). Some of Holub’s examples showing the problems with inheritance rely on the fact that overriding takes place (Holub 2003). One of the issues addressed by Steyaert et al.’s reuse contracts is to deal with situations involving overriding (Steyaert et al. 1996). Overriding is also a feature of the fragile base class problem.
(Mikhajlov & Sekerinski 1998). The presence of overriding may not be the only source of problems, but it does seem to be a frequent concern, and so worth studying.

2.2 Related Work
Lorenz and Kidd (Lorenz & Kidd 1994) proposed “number of methods overridden” (NMO) as a metric in 1994, however only a few, small, studies using it have been reported.

Briand et al. carried out a study to investigate the relationship between measurements from different metrics and the probability of fault detection in classes during testing (Briand et al. 2000). They looked at 8 C++ systems developed by students, a total of 113 non-library classes. NMO was one of the metrics they investigated. Their results indicated that fault-proneness was correlated to NMO. Of interest to us is to what degree overriding was present in the systems they examined. The largest NMO measurement they observed was 10.

El Emam used the NMO metric as part of an analysis of the effect that class size had on a number of object-oriented metrics (Emam et al. 2001). However due to the fact that they only had 6 non-zero measurements these were excluded from their analysis.

There have been a number of studies using other object-oriented metrics, including depth in inheritance tree (DIT). These have not been directly investigating overriding, but the following are of interest.

An experiment by Daly et al., where the subjects were timed modifying C++ systems with varying levels of inheritance (zero, three and five levels), revealed that a system with three levels of inheritance was easier to maintain than the equivalent system with zero or three levels of inheritance (Daly et al. 1996).

The Daly et al. study was later replicated, with the results suggesting the opposite effect — that inheritance had a positive effect on maintenance time (Cartwright 1998). Harrison et al. also replicated the experiment and found that the system with zero inheritance was easier to modify than the equivalent systems with three or five levels of inheritance (Harrison et al. 2000). That these studies could get inconsistent results suggests that depth of inheritance is not the only variable that needs to be controlled. We suggest the amount of overriding is something that also needs to be accounted for.

Tempero et al. studied inheritance structures of 93 of the 100 applications we examined in our study (Tempero et al. 2008). That study used a wider interpretation of “inheritance” than we, including, for example, classes that implement interfaces. The study did separate out measurements for implementation inheritance, which indicated that implementation inheritance was not at all rare, and in fact could be quite significant in some applications.

Baxter et al. looked at the distributions of a number of metrics for object-oriented code in 56 Java applications (most of which are in this study) (Baxter et al. 2006). In almost all cases the distributions they observed were one of two classes of distributions — powerlaws and so called “truncated curves” that are almost, but not quite, powerlaws (such as log-normal or stretched-exponential). The Tempero et al. study also noted the same two classes of distributions.

3 Methodology
As noted in the previous section the metric that has been generally used regarding overriding is NMO, the number of superclass methods a subclass overrides. In carrying out our study, we quickly found there are many different characteristics of overriding that we wanted to examine, leading us to develop many new metrics. For this particular study, we defined and used more than 25 metrics, and there are more still we could have considered. We do not have space to present all these results, and so rather than listing every potential metric, we motivate the need for multiple metrics here, and then define a practical set of metrics. The next section presents the result of these metrics applied to our Java corpus.

Before discussing metrics, we need to be specific about the entities the metrics are to apply to and the attributes we wish to measure (Fenton & Pfleeger 1998). NMO is typically defined in terms of “classes” and “methods.” Java, however, has many “class-like” things, such as “interfaces”, and its “methods” may or may not have implementations, so it is not always clear what NMO means. There are several possible decisions we could make. As this is our first study of overriding, we have chosen the simplest set we thought would be useful, as we now describe.

We want to measure how overriding is actually used, and to do so we must be precise about what we mean by “overriding”. For this study, we focus on overriding that results in replacement of implementation, as that is a common theme in the arguments against the use of inheritance. This means that we do not consider abstract methods, as there is no implementation being replaced in such cases. There is also no choice regarding abstract methods — they must be “overridden”. The decision to make a method abstract seems to be different in nature to the decision to replace an implementation, and so we prefer to measure those decisions separately (however see the discussion regarding “trivial” overriding in section 4.5). Note that this means our notion of overriding is not exactly that from the Java Language Specification (Gosling et al. 2005, 8.4.8).

Thus, for our study, a method m in class B overrides a method from class A, an ancestor of B, if it overrides in replacement of implementation, as that is a common theme in the arguments against the use of inheritance. This means that we do not consider abstract methods, as there is no implementation being replaced in such cases. There is also no choice regarding abstract methods — they must be “overridden”. The decision to make a method abstract seems to be different in nature to the decision to replace an implementation, and so we prefer to measure those decisions separately (however see the discussion regarding “trivial” overriding in section 4.5). Note that this means our notion of overriding is not exactly that from the Java Language Specification (Gosling et al. 2005, 8.4.8).

Our definition is illustrated in figure 1. In this example the implementation a1() in B overrides a1() inherited from A, however there is nothing as no implementation of that method is inherited from A. Note that a1(int) in B is not involved in any overriding, being an example of overloading — a1() and a1(int) are two different methods with different signatures, although they share the same name. Since constructors and static and private methods cannot be overridden, they are not considered in our metrics. Synthetic methods are not considered as they are not written by developers. We do include native methods. It is debatable whether we should include them, however overriding a native method (or vice versa) is still replacing implementation, and we do not expect this to happen so much as to significantly change our results one way or the other.

We want to try to characterise the decision to override a method. The metrics we have in mind are similar to the definition of classes from the standard library or third-party libraries, but does have some control over other user-defined classes, consequently we believe this decision is different. As we do not have space to show both, in this paper we limit the results to just overriding from other user-defined classes.

As we are interested only in implementation, we do not consider interfaces (and annotations). While
exceptions are in some ways just like any other class syntactically, they are treated specially by the Java run-time, and have quite a different role from other “normal” classes. We feel that for the moment we will get a clearer picture of how developers use implementation inheritance if we ignore exceptions. For similar reasons, we count inner classes, but we ignore generic classes, the super box contains all methods belonging to the super class, that fact will not be represented in our measurements.

As we do not consider classes from the standard library or third-party libraries, an issue arises in that for any user-defined class that extends such classes, we cannot determine what is actually being inherited. For this reason, we only consider methods inherited from other user-defined classes, that is, if (for example) a class overrides a method from the standard library, that fact will not be represented in our measurements.

To summarise: we measure only user-defined classes, including abstract classes and all types of inner classes; and methods (e.g. the number of overriding methods |O|), or proportions of one set with respect to another (e.g., the proportion of overriding methods to those declared |O|/|D|) or the proportion of methods replaced |R|/(|I| + |R|)). These metrics have classes as their entities (that is, a measurement is for a single class).

We generalise figure 2 to define metrics over all classes and all methods in applications (that is, taking whole applications as entities and measuring classes and methods). For the classes in applications metrics, the top (super) box contains all classes that have subclasses and the bottom (sub) box has all subclasses. Classes are then categorised as to whether or not they contain methods that have implementations come from P, or declared, meaning their implementations are defined in C’s definition. The declared methods can be further divided into overriding, meaning their implementations replace those in P, or new.

To distinguish metrics related to parent and child classes, we use the term “replaced” for parent methods, rather than “overridden”; we use the term “overriding” only for child methods. For any given pair of parent and child classes the number of replaced methods must be the same as the number of overriding methods — the alignment in the figure is intended to indicate this. For a given parent class, the sets I and R may differ for different child classes.

Several metrics relating to overriding can be derived from figure 2, such as sizes of the various sets (e.g. the number of overriding methods |O|), or proportions of one set with respect to another (e.g., the proportion of overriding methods to those declared |O|/|D|) or the proportion of methods replaced |R|/(|I| + |R|)). These metrics have classes as their entities (that is, a measurement is for a single class).

To summarise: we measure only user-defined classes, including abstract classes and all types of inner classes; and methods (e.g. the number of overriding methods |O|), or proportions of one set with respect to another (e.g., the proportion of overriding methods to those declared |O|/|D|) or the proportion of methods replaced |R|/(|I| + |R|)). These metrics have classes as their entities (that is, a measurement is for a single class).

We generalise figure 2 to define metrics over all classes and all methods in applications (that is, taking whole applications as entities and measuring classes and methods). For the classes in applications metrics, the top (super) box contains all classes that have subclasses and the bottom (sub) box has all subclasses. Classes are then categorised as to whether or not they contain methods that have implementations come from P, or declared, meaning their implementations are defined in C’s definition. The declared methods can be further divided into overriding, meaning their implementations replace those in P, or new.

To distinguish metrics related to parent and child classes, we use the term “replaced” for parent methods, rather than “overridden”; we use the term “overriding” only for child methods. For any given pair of parent and child classes the number of replaced methods must be the same as the number of overriding methods — the alignment in the figure is intended to indicate this. For a given parent class, the sets I and R may differ for different child classes.

Several metrics relating to overriding can be derived from figure 2, such as sizes of the various sets (e.g. the number of overriding methods |O|), or proportions of one set with respect to another (e.g., the proportion of overriding methods to those declared |O|/|D|) or the proportion of methods replaced |R|/(|I| + |R|)). These metrics have classes as their entities (that is, a measurement is for a single class).

We generalise figure 2 to define metrics over all classes and all methods in applications (that is, taking whole applications as entities and measuring classes and methods). For the classes in applications metrics, the top (super) box contains all classes that have subclasses and the bottom (sub) box has all subclasses. Classes are then categorised as to whether or not they contain methods that have implementations come from P, or declared, meaning their implementations are defined in C’s definition. The declared methods can be further divided into overriding, meaning their implementations replace those in P, or new.

To distinguish metrics related to parent and child classes, we use the term “replaced” for parent methods, rather than “overridden”; we use the term “overriding” only for child methods. For any given pair of parent and child classes the number of replaced methods must be the same as the number of overriding methods — the alignment in the figure is intended to indicate this. For a given parent class, the sets I and R may differ for different child classes.

Several metrics relating to overriding can be derived from figure 2, such as sizes of the various sets (e.g. the number of overriding methods |O|), or proportions of one set with respect to another (e.g., the proportion of overriding methods to those declared |O|/|D|) or the proportion of methods replaced |R|/(|I| + |R|)). These metrics have classes as their entities (that is, a measurement is for a single class).

We generalise figure 2 to define metrics over all classes and all methods in applications (that is, taking whole applications as entities and measuring classes and methods). For the classes in applications metrics, the top (super) box contains all classes that have subclasses and the bottom (sub) box has all subclasses. Classes are then categorised as to whether or not they contain methods that have implementations come from P, or declared, meaning their implementations are defined in C’s definition. The declared methods can be further divided into overriding, meaning their implementations replace those in P, or new.

To distinguish metrics related to parent and child classes, we use the term “replaced” for parent methods, rather than “overridden”; we use the term “overriding” only for child methods. For any given pair of parent and child classes the number of replaced methods must be the same as the number of overriding methods — the alignment in the figure is intended to indicate this. For a given parent class, the sets I and R may differ for different child classes.

Several metrics relating to overriding can be derived from figure 2, such as sizes of the various sets (e.g. the number of overriding methods |O|), or proportions of one set with respect to another (e.g., the proportion of overriding methods to those declared |O|/|D|) or the proportion of methods replaced |R|/(|I| + |R|)). These metrics have classes as their entities (that is, a measurement is for a single class).

We generalise figure 2 to define metrics over all classes and all methods in applications (that is, taking whole applications as entities and measuring classes and methods). For the classes in applications metrics, the top (super) box contains all classes that have subclasses and the bottom (sub) box has all subclasses. Classes are then categorised as to whether or not they contain methods that have implementations come from P, or declared, meaning their implementations are defined in C’s definition. The declared methods can be further divided into overriding, meaning their implementations replace those in P, or new.

To distinguish metrics related to parent and child classes, we use the term “replaced” for parent methods, rather than “overridden”; we use the term “overriding” only for child methods. For any given pair of parent and child classes the number of replaced methods must be the same as the number of overriding methods — the alignment in the figure is intended to indicate this. For a given parent class, the sets I and R may differ for different child classes.

Several metrics relating to overriding can be derived from figure 2, such as sizes of the various sets (e.g. the number of overriding methods |O|), or proportions of one set with respect to another (e.g., the proportion of overriding methods to those declared |O|/|D|) or the proportion of methods replaced |R|/(|I| + |R|)). These metrics have classes as their entities (that is, a measurement is for a single class).

We generalise figure 2 to define metrics over all classes and all methods in applications (that is, taking whole applications as entities and measuring classes and methods). For the classes in applications metrics, the top (super) box contains all classes that have subclasses and the bottom (sub) box has all subclasses. Classes are then categorised as to whether or not they contain methods that have implementations come from P, or declared, meaning their implementations are defined in C’s definition. The declared methods can be further divided into overriding, meaning their implementations replace those in P, or new.

To distinguish metrics related to parent and child classes, we use the term “replaced” for parent methods, rather than “overridden”; we use the term “overriding” only for child methods. For any given pair of parent and child classes the number of replaced methods must be the same as the number of overriding methods — the alignment in the figure is intended to indicate this. For a given parent class, the sets I and R may differ for different child classes.
tion, counting nested classes as individual classes.

3.2 Population Measured

We base our empirical study on the Qualitas Corpus (Qualitas Research Group 2008) a collection of open source Java applications we have been using for similar studies. The version we used was released on 3 June 2008 and consists of 100 distinct applications, including multiple releases of some applications. The details, including the full list of applications studied, is available on the Qualitas Corpus website. The measurements were performed on the bytecode representation of the applications. While this is not a true representation of the source code, all information regarding class relationships and method declarations is available from bytecode, so this is sufficient for our study.

4 Results

4.1 Application Measurements

We begin by looking at measurements for applications. Figure 4 shows the number of classes that override something ([CO]) along with the number of classes that inherit something (NCI) and the number of classes in the application (NC). The chart has been truncated in the y-axis to make the scale clearer. It shows the distribution of the population we studied. There are 24 applications with more than 1000 classes and 5 with more than 2000 classes (these shown in table 1). NCI and [CO] both roughly (but not exactly) follow NC in that the larger applications tend to inherit and override more.

Figure 5 shows the number of classes that inherit from other user-defined classes as a proportion of user-defined classes (that is NCI/NC). This is the UDCCDU metric used by Tempero et al (Tempero et al. 2008). As we will do in other cases, the x-axis is the rank of an application for a given size metric, rather than its size. There are many applications with similar sizes, meaning plotting against size would make the individual measurements difficult to distinguish. The applications are ordered according to the appropriate size metric (NC in this case) in order to see if there is a trend with respect to size.

Figure 5 is useful to help evaluate the later results, but it is also useful to see the degree to which developers have tried to inherit implementation. Of the 100 applications, 31 have at least half of their classes inheriting from other user-defined classes (recall that this does not include classes that inherit from standard library or third-party classes). The minimum is 4% ([mvnforum]), the maximum is 85% (jparse), and the median is 38% (drawswf). There is no obvious trend with respect to size; the right-most 24 (more than 1000 classes) measurements are not at all different to the left-most 76.

4.2 Classes that Override

Figure 6 shows the proportion of classes that have at least one method that overrides another method to the number of classes that could possibly have such methods, that is, the number of classes that inherit from some other user-defined class. While there are some applications that have low proportions, they are applications that do not have many classes that inherit from other classes. The minimum is 0% (jasm1, NCI=21), median is 53% (trove, NCI=125), and maximum is 100% (nekohtml, NCI=6), which also has the smallest number of user-defined classes. The application with the largest number of classes is eclipse (17621), which is at 67%. The next largest is jre (11181) with 56% of its classes that inherit overriding at least one inherited method. The application with the second-largest proportion is ireport, which has 112 classes that inherit, but its proportion of classes that inherit is relatively small (112/1232, 8%). Again there is no obvious trend with respect to size. We also looked at whether the degree to which inheritance is used is related to how much overriding takes place and found no obvious patterns.

We can also consider how often, on average, a class overrides methods. We start with a simple average over all classes in an application. Figure 7 shows two averages — one is the average number of methods overridden over all classes that inherit something ([MR]/NCI, circle) while the other shows the average number of methods overridden over those classes that override something ([MR]/[CO], squares). There is one application (nekohtml) where the two averages are the same, indicating that every

<table>
<thead>
<tr>
<th>Application</th>
<th>[CO]</th>
<th>NCI</th>
<th>NC</th>
</tr>
</thead>
<tbody>
<tr>
<td>gt2</td>
<td>900</td>
<td>1436</td>
<td>2515</td>
</tr>
<tr>
<td>jtopen</td>
<td>374</td>
<td>931</td>
<td>2605</td>
</tr>
<tr>
<td>azureus</td>
<td>409</td>
<td>1261</td>
<td>4159</td>
</tr>
<tr>
<td>jre</td>
<td>3240</td>
<td>5828</td>
<td>11181</td>
</tr>
<tr>
<td>eclipse</td>
<td>6042</td>
<td>8983</td>
<td>17621</td>
</tr>
</tbody>
</table>

Table 1: The five largest applications studied, as measured by NC.
class that inherits overrides something. This application also has the largest average (9.67), however there are only 6 classes that inherit anything. The median MR|NCI is 1.14, 42 applications have an average of 1 or less, 77 applications have an average of 2 or less, and 93 have an average of 3 or less. The largest application, eclipse, has an overall average of 1.91, that is, for every class in eclipse that inherits from another class in eclipse, on average nearly two of its methods are overriding.

While there are a number of applications with a low degree of overriding, there are also those where it is quite high. This suggests that the way overriding is used might be due to either the domain or the programming style (also see Section 4.4). For any subclass in an application, there is likely to be at least one method that overrides an inherited method.

4.3 Methods that are Overridden

In the results we have presented so far we have considered a class to “use overriding” if it has one method that overrides another. It may be that the same method is overridden many times (when its class has many descendants). To get an overall view of this, Figure 8 shows the number of inherited methods in an application that are overridden by something else as a proportion of the total number of inherited methods (MR|NMI).

While there are some high values (the highest is 82% — nekohtml again), the median is 18% (hibernate — 509 methods overridden out of 2822 inherited), the smallest is, unsurprisingly, jasml with 0%. The application that overrides anything with smallest value is informa at 2% (4 out of 171 methods inherited) and overall 75% of the applications have 25% or fewer methods that get overridden.

4.4 Longitudinal Results

Another view of the data is a longitudinal one. We tracked the measurements of each of the releases we have for jgraph (29 releases), freecol (12), azureus (11), and eclipse (13). Figure 9 shows the results for the proportion of classes that have a method that overrides to the classes that inherit (CO|NCI). In interpreting this figure, it must be noted that the releases of jgraph come from a shorter period of its entire lifespan than the other applications, as the earliest release we have is relatively late in its development compared to the other applications, which would explain the fairly straight line. While jgraph is both the smallest of the 4 and has the highest measurements, the largest application, eclipse, has the next highest measurements. The other two applications go through significant changes.

Figure 10 shows the proportion of the overridden methods to the total number of methods inherited (MR|NMI). Here we see quite similar curves, although not close enough to be a common trend. The relatively flat nature of all curves indicates that as applications grow in size (as they all mainly did over
the releases we studied) the number of overridden methods also grows at the same rate, although figures 9 suggests the distribution of such methods to classes changes. This is further evidence that the use of overriding is related to either domain or programming style. Determining which is the case would be an important result.

4.5 Presence of Trivial Implementations

When manually examining code, we noticed a number of cases of overriding where the bodies of the methods were either empty, or returned a constant, which we call “trivial” implementations. For example, the abstract class org.apache.tools.ant.Task has methods that have empty bodies (such as init and execute) that are overridden by inherited classes. While it might make sense that init has a default implementation that does nothing, it is not clear why execute should, and in fact execute is overridden by every subclass. Since Task is abstract, there is no obvious reason why execute could not be made abstract. Classes with empty implementations that are overridden may or may not make sense, but identifying such cases may provide opportunities for changing design choices (or at least improving documentation).

There are also cases where both the overridden and the overriding methods are empty. For example the init method in Task is overridden by an empty implementation in org.apache.tools.ant.taskdefs.optional.sound.SoundTask. While this seems harmless (especially in this case) it is still a questionable situation that deserves clarification.

More troubling is when a non-trivial method has been overridden by an empty method. For example, org.apache.tools.ant.DirectoryScanner has methods (e.g. AddDefaultExcludes) that have non-trivial implementations that are overridden in org.apache.tools.ant.types.optional.depend.DependScanner by empty implementations. This throws away inherited implementation, which seems very questionable. In principle, this situation can be removed by having a common (possibly abstract) superclass to DirectoryScanner and DependScanner containing the shared implementation and then each class providing their specific (possibly trivial) implementation. Identifying such situations indicate possibilities for refactoring.

We measured how many methods there were in an application that had empty bodies but overrode methods with non-empty bodies, and how many classes had such methods. The results are shown in figure 11, where the number of classes are shown as boxes, and the number of methods as circles. As can be seen, for most applications the numbers were fairly small, with 31 applications having no such methods.

The figure also suggests that there is often only one such method per class, which is the case for 57 applications. The maximum number of classes is in eclipse (231, 386 methods overall) whereas the maximum number of methods is jre (402 in 215 classes). The degree to which such methods exist is not completely related to the amount of overriding with compiere (424 classes with overriding, 8th most) having just 1, and weka (445 classes with overriding, 5th largest) having 6.

Our conclusion is that there is a significant degree of this questionable use of overriding. These results suggest that it is worthwhile providing IDE support to identify when this kind of use occurs and more refactoring support to help remove it.

4.6 Class Measurements

We now turn to the measurements for individual classes. Here our main interest is in the shapes of the distributions, rather than individual values. As we cannot show all distributions of all applications, we have picked two representative applications. Figures 12 and 13 show results for jgroups and eclipse side-by-side. Eclipse is the largest application we studied (NC=17621) whereas jgroups is relatively small (924, although it is the 75th largest in the corpus by NC). Points to note are:

- The first column of figure 12 shows the Number of Methods Declared (|D|). This is a almost-but-not-quite-power-law (truncated curve) as observed previously (Baxter et al. 2006).
- The next column shows Number of Methods Overridden (|O|), again showing truncated curves.
- The third column shows the Number of Methods Inherited (NMC).
- The last column shows Number of Methods (NMC). Recall that this metric considers only non-private, implemented methods in a class, but includes both those declared in the class and those inherited (but not overridden).
- In figure 13, the top row shows how many classes have a given proportion of methods that override to the methods that are inherited (|O|/NMC). As Figure 7 indicates, for most applications, when a class overrides something, it typically overrides only 1 method. Yet the two charts shown are by no means unique. Of particular
Figure 12: Log-log frequency distributions of measurements for jgroups (top) and eclipse (bottom). X-axis left to right: Number of Methods Declared in a Class (|D|), Number of Methods Overridden in a Class (|O|), Number of Methods Inherited in a Class (NMC), Number of implemented Methods in a Class (NMC).

interest is the number of classes in jgroups that override all of the methods inherited (38 of 273 classes that inherit something, or 14%). This is quite different to eclipse, but again not unique.

• The bottom row of Figure 13 shows how many classes have a given proportion of methods that override of the methods that are declared (|O|/|D|). Of note here is the number of classes in both applications where all of the methods declared in a class are overriding an inherited method although the proportion of classes in this category are quite different (for jgroups, it is 39 out of 273, or 14%, and for eclipse, it is 2955 out of 8983, or 33%).

We now focus on the classes that either override all inherited methods (Figure 14) or declare only methods that override inherited methods (Figure 15).

Figure 14 shows the number of classes that override all inherited methods as a proportion of the number of classes that inherit something. The application with the largest proportion of such classes is galleon (19%). The application with the largest absolute number of such classes is in fact the second-largest application, jre (176 classes that override all inherited methods, where NCI=5828, or 3%).

Classes that override all inherited methods get little benefit from implementation inheritance. It is possible that in some cases they benefit through the use of super but such large numbers might be due to the fact that there are many classes that don’t inherit many methods (e.g. only 1 method is inherited) but this is an area that deserves further study.

We looked at some of the classes that override everything that was inherited, concentrating on those classes that inherited 10 or more methods. There were 9 such applications with jre having 11 classes, poi 4, eclipse 2, and javacc, jspwiki, c_jdbc, drjava, jruby, and gt2 one each.

• org.javacc.jjdoc.HTMLGenerator 18 methods inherited from org.javacc.jjdoc.Generator. This is the only class that inherits from its parent, which provides text generation. These two classes could easily be redesigned to implement a common interface.

• com.ecyrd.jspwiki. TranslatorReader$HTMLRenderer 36 methods inherited from com.ecyrd.jspwiki. TranslatorReader$TextRenderer. Comments in the source indicate that further work is intended with respect to the design of this class.

Again this class and the class that it inherits from could implement a common interface.

•javax.swing.plaf.synth.ImagePainter, javax.swing.plaf.synth.

ParsedSynthStyle$DelegatingPainter 113 methods inherited from javax.

swing.plaf.synth.SynthPainter, an abstract class. In this case these aren’t the only subclasses of SynthPainter. com.sun.java.swing.plaf.gtk.GTKPainter also inherits. But again a common interface seems possible.

This sample suggests that those classes that override everything they inherit, and the other classes that participate in the same inheritance hierarchies, are candidates for refactoring by introducing appropriate interfaces.

Figure 15 shows the number of classes that declare only methods that override something. The largest is freecs with 87% and the median is jgrapht (%15). This is clearly a much more common situation.

4.7 Depth of Inheritance

We might expect that classes deeper in the inheritance hierarchy are bigger than those at the top. To investigate these claims we looked at the average size of classes at different levels of inheritance. Figure 16 shows the average values for NMC, NMC, |O|, and |D| for each application for DITCCUD=1 and DITCCUD=5. We chose DITCCUD=5 because at deeper levels of inheritance we have fewer classes and so trends are not as clear as they are at DITCCUD=5, although they are consistent with what we show. For comparison, we also show the average DITCCUD per application for those classes that inherit from user-defined classes (and so the DITCCUD is at least 1).

The first thing to note is that the average depth of inheritance trees seems largely unrelated to the size of the application (as measured by NCI). While the maximum depths increase with application size, the number of shallow classes also increases with size.

The NMC measurements confirm the expectation that, on average at least, the number of implemented methods for a class increases with depth in the inheritance tree. It is interesting that this is more obvious with the larger applications, despite the fact that their average DITCCUD values are no bigger than the smaller applications.
Figure 13: Frequency distributions of measurements for jgroups (left) and eclipse (right). Proportion of methods overridden to methods inherited (|O|/NMIC — top) and Proportion of methods overridden to methods declared (|O|/|D| — bottom).

Figure 14: Proportion of classes that inherit of classes that override all inherited methods.

Figure 15: Proportion of classes that inherit with all declared methods overriding something.

The number of methods inherited (NMIC) are also larger at the deeper levels, which is not surprising as if there are more methods at deeper levels then there are more methods to inherit. The number of methods overridden is not so conclusive. There is one large application (jtopen NC 2605) that has a high average, but that one value is not enough to indicate a trend. Note that the scale for the |O| measurements is significantly smaller than for NMC and NMIC.

The number of methods declared (|D|) does not appear to be any different at different depths in the inheritance tree. In fact, generally |D| is smaller at deeper levels, which is what we would expect. The fact that the number of methods declared at deeper levels is anything like the number at the top of the inheritance tree is probably due to the number of overridden methods, which |D| includes.

5 Discussion

For the question “how much overriding is there” we can say this: half the applications have 53% or more of their subclasses using overriding (figure 6), and 58 applications average more than one method overridden per subclass (figure 7). These results are largely independent of the size of the application. In other words, anyone dealing with a subclass has a better than even chance of having to also deal with overriding. Furthermore, this is overriding of methods written for the application. We do not include overriding of third-party or standard library methods, which clearly also takes place. If overriding is a source of difficulty then what we have observed suggests it would contribute a non-trivial cost.

Of course the ability to override does have some benefits, and so this cost may be justified by the benefits. While we have not specifically looked at the benefits accrued by overriding, we noted two prominent characteristics of actual overriding whose benefits are very questionable. These are:
Figure 16: Average measurements for each application in NCI order for (left to right, top to bottom) NMC, NMIC, |O|, and |D| for classes with DITCCUD 1 and 5. The average and maximum DITCCUD measurements are shown for each application for comparison. Lines are used to emphasise trends.

- The number of cases where either the overridden or overriding method is trivial. We believe that when this occurs, there is the possibility that the inheritance structure can be refactored using interfaces (or at least abstract classes) to remove this use of overriding. In particular, if the overriding method is trivial (and especially if the overridden method is also trivial) there is significant room for improvement.

- The number of cases where all inherited methods are being overridden. In such cases, there appears to be no benefit due to implementation inheritance, and so refactoring again seems possible to remove the overriding.

Given the results relating number of overriding methods to fault proneness (Briand et al. 2000), it would seem a good first start would be to remove as much of this trivial overriding as possible.

IDEs such as Eclipse already give an indication (typically a decoration in the editor) when overriding occurs. This means the infrastructure exists to be able to signal to developers when these cases have occurred, allowing proactive assessment. Further, we believe that these observations can be exploited to provide automated refactoring support, that is, in some cases it should be possible to automatically determine how to restructure the inheritance hierarchy to remove overriding.

Figure 12 suggests we should treat averages with caution. Whether or not the number of methods overridden in a class (|O|) is a powerlaw, it is certainly highly skewed. While most classes that override anything override only a few methods, there are classes that override many methods (up to 100 in the case of eclipse). We need to better understand why so many methods need to be overridden. Some cases can be explained by the provision of default implementations for large interfaces. For example, the visitor pattern when used for parsers results is very large interfaces. However this cannot explain all measurements we see.

Another question we need to consider is, should there be more overriding? Is it the case that the advice of people such as Gamma is being followed and, with less use of inheritance, there is less opportunity to exploit the benefits of overriding? Answering this question requires better characterisation of benefits of overriding than exists, and we feel developing such a characterisation ought to be a priority if we are to use inheritance to its fullest benefit.

One possibility that we have to consider is if alternative mechanisms to inheritance, and hence to overriding, were used instead. In C++, the friend facility was often used to circumvent the need to re-engineer the inheritance hierarchy (even in a relatively small way); empirical studies have show that friends are indeed used frequently and may also be the cause of possible faults (Counsell & Newson 2000, Briand et al. 1997). Our study did not consider Java interfaces or abstract methods in its analysis. One suggestion could be that the use of interfaces may be the mechanism through which complex inheritance relationships (even multiple inheritance as we know it in C++) are avoided.

5.1 Threats to Validity

The main threat to the validity of our conclusions is the corpus we used, which consists entirely of successful open-source Java applications, many of small to medium size. Our results do apply to at least these applications, many of which (e.g. eclipse, ant) are some of the most used Java programs worldwide, and so we are confident that our conclusions hold within this population. We do not know whether or not the level of overriding is different for applications created under different development models than open source, or in different languages, or in applications that are unsuccessful. More empirical studies are needed to determine this.

6 Conclusions

We have presented a large empirical study on the use of overriding in Java open source software, considering 100 applications with over 100,000 user-defined types. We have developed a general approach to investigating overriding, and a set of metrics for measuring different aspects of overriding. We have found substantial use of overriding — every subclass in an
application is likely to override at least one inherited method. We also discovered two common uses of overriding that are questionable, namely when:

- classes override every inherited method, and
- when classes provide empty implementations that override non-empty implementations.

We believe these two situations represent unnecessary overriding, and further, are amenable to automatic detection and support for removal. Our experience is that these kinds of findings are only possible through large-scale empirical studies.

This is by far the largest study of this kind. It provides a benchmark of the level of overriding used in this population that is typical today. For example, our results can be used to identify suitable candidates in the corpus for study on the effects of use of overriding on development or maintenance costs. This benchmark can also be used to evaluate the use of overriding in future studies, both in Java open source software and in other languages supporting software.

There are a number of directions future research could take. We only considered overriding methods from other user-defined classes in this study. We need to also examine how overriding is used with respect to the standard library and third-party libraries. We have begun looking at how super is used. Its use could be the source of difficulty with respect to overriding, as indicated by the so-called “Yo Yo” problem (Binder 1999).

Our study needs to be replicated by others to verify our results. Other populations, such as commercially-developed Java applications and applications developed in other languages, need to be studied. We have noted our corpus consists of what we have called “successful” applications. It would be very interesting to see if “unsuccessful” applications (particularly those that have been unsuccessful due to difficulty in maintaining the code) exhibit significantly different levels of overriding.

The benefits overriding provides needs to be better characterised, or at least more indicators of questionable use of overriding determined. For example, we have noted occurrences of overriding where the replacement implementation is identical to what it is replacing (trivial, in both cases). Does it happen that identical non-trivial overriding takes place? It seems unlikely, but we won’t know for sure until we look. It is only through empirical studies of the kind we have presented that such questions can be settled.

Acknowledgements

Most of this work was done while Tempero was a visiting researcher to the BESQ project at Blekinge Institute of Technology, Ronneby, Sweden, whose support he gratefully acknowledges.

References


