Subclass Instantiation Distribution

Amy Wheeler and Dave Binkley
Loyola University Maryland
Baltimore MD, 21210, USA
awheeler1@loyola.edu, binkley@cs.loyola.edu

Abstract

During execution, an objected-oriented program typically creates a large number of objects. This research considers the distribution of those objects that share a common superclass. If this distribution is uniform, then all subclasses are equally likely to be instantiated. However, if not, then the lack of uniformity can be exploited by giving preferential treatment to the dominant class (or classes). For example, a tester might spend greater testing resources on the dominant class while an engineer refactoring the code might begin with a more dominant class. An experiment designed to investigate the distribution of subclass instantiations was performed using eight Java programs containing almost half a million lines of code and just over three thousand classes. The results show that outside a few infrequent instances, most distributions are heavily skewed.

1 Introduction

This paper investigates the dynamic distribution of subclass instantiations. A better understanding of this distribution is advantageous to, among others, developers and managers. For example, consider the situation in which one of an object’s subclasses is instantiated significantly more often than its peers. In this case, a developer seeking to increase performance, or a manager seeking to focus testing effort before an imminent release date, can exploit distribution information by focusing on the heavily instantiated class. Other consumers of this information include maintainers, testers, tools, and tool builders. For example, maintainers benefit from knowing where to focus their initial comprehension effort, while better informed static analysis tools can focus their analysis on particularly common parts of the distribution.

In the absence of distribution information, any analysis must consider all possibilities. These possibilities range from a completely uniform distribution, where all subclasses are equally likely to be instantiated, to a heavily skewed distribution in which only one subclass is ever instantiated. Knowledge about the distribution can range from specific dynamic information to general static information. At the one end, dynamic distribution information obtained from a prior execution of a program has several immediate uses. For example, with the aim of increasing performance, heavily used classes could be favored with advantageous memory placement or additional resources, such as registers by the JIT [10]. Alternatively, testers might focus more attention on those classes with a higher frequency in the distribution.

While dynamic distribution information is of clear value, static knowledge about distributions is also useful as it can guide tasks such as tool development. Assume that all distributions are heavily skewed and thus have one or a few dominant subclasses. This makes it worthwhile for an aggressive compiler to inject code aimed at the runtime determination of these dominant classes. Once identified, such classes can be given favored treatment (for example, more frequent attention by the garbage collector [18]). Furthermore, if the skew is found to correlate with static metrics, such as lines-of-code or cohesion [6], then a compiler writer could use these metrics to exploit the skewed distribution by, for example, giving favored treatment in the optimizer. Thus, statically knowing the expected distribution supports improved static analysis.

The primary contribution of this paper is an empirical understanding of the distribution of subclass instantiations through an analysis of almost half a million lines of Java code from over three thousand declared classes. This empirical work is presented as follows: first, Section 2 describes the tool used to extract the distributions and the statistical techniques used to analyze the data. Then, Section 3 presents the empirical study’s design and Section 4 its results. Section 5 illustrates how existing techniques can exploit the empirical results. Then, Section 6 considers related work and Sections 7 and 8 provide some directions for future work and a summary of the paper.

2 Background

This section provides background information on aspect-oriented programming (AOP), the AOP tool used, JBoss, and the statistical techniques used. To capture the dynamic object creation data without recompiling requires instrumenting the bytecode. Aspect Oriented Programming [11, 13] is well suited to this task as it can be used to inject bytecode at the point of object instantiation.

Aspects are crosscutting concerns woven into the base code. Example crosscutting concerns include persistence, transaction management, and contract enforcement [12, 3]. An aspect is composed of one or more pieces of advice that
are woven into the base code by intercepting the execution flow, without any need for the base code to be aware of the aspect. Thus, the base code remains oblivious to the functionality added by the advice of an aspect [8].

Join points are points in a program’s execution where advice can be inserted (woven in). They include method calls, constructor executions, and field updates. In general, advice can be woven in before, after, or around (in place of) a join point.

Finally, pointcuts denote sets of join points and are specified using pointcut designators. Only a small subset of the many pointcut designators were required to gather the instance creation information. For intercepting constructor calls, the most general pointcut (i.e., the one leading to the largest set of joint points) is "construction(->new(..))", which describes any constructor from an arbitrary class (using the class expression "*"*) with an arbitrary number of formal parameters (the "(..)"). Point-cut designators can be combined using standard boolean operators such as and and not (denoted "!").

Advice is specified as a Java method. The particular AOP system used, JBoss¹, weaves in advice at the bytecode level. This obviates the need to recompile (or even have access to) the source code. JBoss uses the javaagent interface of the JVM to modify the class loader before main() is called. The modification registers a ClassTransformer, which supports the interception of calls from the class loader where the bytecode of a loaded class can be rewritten. The use of the javaagent interface means that no weaving is done into classes loaded before this interface. This affects core classes (e.g., java.lang.*), but not libraries or jar files loaded after the JVM registers the class transformer.

JBoss uses an xml configuration to specify pointcuts. Figure 1 shows the xml file used to instrument constructor invocations. The xml code binds the interceptor, named ConstructorInterceptor, from the aspect (class) Tracer, to constructors matching the specified pointcut. In the figure, this set of join points includes all constructor calls except those within the tracer itself. As explained in Section 3.3, two programs required smaller sets of join points. The pointcut also specifies that the advice is to be woven in before each join point.

Figure 1 also shows the advice that records each class created along with its chain of superclasses. While a chain cannot start with an abstract class, abstract classes can be included in a chain. The final action of the advice, invokeNext(), executes the original constructor. Post execution, the recorded information is processed to obtain the object instantiation distributions.

Finally, this section introduces the statistical techniques used to analyze the distributions. The first test used is a \( \chi^2 \) test of goodness of fit. With a null hypothesis that all outcomes are equally likely to occur, rejecting the null hypothesis (finding \( p < 0.05 \)) indicates a skewed distribution.

¹http://labs.jboss.com/jbossaop

---

```
jboss-aop.xml
<aspect class="Tracer" />

<bind pointcut="construction(->new(..))" and !construction(Tracer*->new(..))"
     before aspect="Tracer"
     name="ConstructorInterceptor"/>
</bind>
</aop>

Tracer.java – shown without error checking code
import org.jboss.aop.joinpoint.*;
import org.jboss.aop.advice.*;
public class Tracer
{
    public Object ConstructorInterceptor
        (ConstructionInvocation ci)
    {
        Class c = ci.getConstructor()
            .getDeclaringClass();
        Chain invocation = new Chain();
        while (c != null)
        {
            invocation.addClass(c);
            c = c.getSuperclass();
        }
        return ci.invokeNext();
    }
}
```

Figure 1. An example aspect and the xml code used to describe the join points where the aspect is woven into the application.

The second technique measures the degree of non-uniformity. This is done by first ranking the subclass instantiation probabilities for a class from highest to lowest and then fitting a curve to the ranked list. The four relevant distributions used in the analysis are illustrated in Figure 2. First, the data for a class with a uniform distribution appears graphically as a horizontal line because, by definition, all subclasses are equally likely to occur. The other three distributions are non-uniform in varying degrees. Since the data is sorted from highest to lowest, all non-uniform distributions are skewed to the right (slope down toward the lower right). The first of the three, a power-law distribution, captures a relation where the frequency of an event varies as a power of some attribute (e.g., its rank in the sorted order of instantiation counts). Power laws characterize a large number of naturally occurring phenomena, such as the inverse-square law governing gravitation attraction, and Pareto’s law of income distribution. In the experiment, the response variable, percent of subclass instantiations, is modeled as \( ke^x \) where \( x \) is the rank of the subclass in terms of the proportion of instantiations. Such models are reported using a triple that includes "P", denoting a power-law
model, $R^2$, the percentage of the response variable’s variation explained by the model, and the exponent $n$, where more negative (larger) values indicate a more extreme skew (i.e., that the first value is more dominant). In a power-law it is rare for $n$ to be less than -3 and as $n$ approaches 0 the distribution approaches a uniform distribution.

The second skewed distribution, an exponential distribution, models percent as $ke^{nx}$. These distributions are reported using the triple “E”, denoting an exponential model, an $R^2$ value, and the scaling factor, $n$. In the analysis, the model with the better fit ($R^2$ value) is reported. In general, the skew is more extreme with a power law as the fit curve initially falls faster than that of the exponential model. This is illustrated in Figure 2.

Finally, a linear decrease, although not as drastic, shares the power-law and exponential distributions’ decreasing likelihood that a particular subclass is instantiated. However, unlike the power-law and exponential distributions, a linear distribution is a straight line. (Exponential distributions are a straight line on semi-log plots, while power-law distributions are a straight line on log-log plots.) The slope of a linear regression line can be used to measure how close to uniform a given distribution is. The smaller the slope of the line, the closer the distribution is to a uniform distribution.

3 Empirical Study Design

This section first presents the three research questions considered. It then describes the eight subject programs studied and the process used to gather the empirical data.

3.1 Research Questions

Research Question 1: Do classes with instantiated subclasses have more than one instantiated subclass?

Research Question 2: If multiple subclasses are instantiated, is the distribution of subclass instantiation uniform or skewed?

Research Question 3: If the distribution is skewed, then does a (a) single or do a few subclasses dominate? (b) higher instantiation count correlate to greater skew?

The rationale behind Research Question 1 is to provide a preliminary check that multiple subclasses are instantiated.

Research Question 2 is the primary focus of the investigation as it considers the existence of skew in the instantiations. Finally, provided skew exists, the rationale behind Research Question 3 is to better understand two attributes of the skew.

3.2 Subjects

Data was collected for the eight programs shown in Figure 3. All were downloaded (mainly from sourceforge) between May 2009 and August 2009. With one exception, jolden, a Java version of the olden benchmarks [4], programs were chosen to be of non-trivial size. Jolden was included to allow comparison with a smaller compute-bound program. The subjects studied represent a convenience sample in that several systems (e.g., the ArtOfIllusion and sandmark) were dropped because they were incompatible with JBoss or did not include a sufficient test suite.

3.3 Process

To extract the instance creation data, each program was first compiled (if necessary, as some were distributed in bytecode form). JBoss was then used to run the application while dynamically weaving in the tracer advice shown in Figure 1.

The pointcut shown in Figure 1 is sufficient for six of the eight programs analyzed. However, with jmeter and GanttProject more restrictive weaving was necessary. This need is often related to private classes or the use of reflection. For jmeter, JBoss was unable to weave before advice into certain static private subclasses due to JVM permission violations and also into classes that used reflection in a manner not compatible with JBoss’s weaving. To work around this, several class subtrees were excluded. These include classes inheriting from org.mozilla.javascript and from org.apache.jorphon. For similar reasons, three GanttProject classes were not compatible with JBoss and thus excluded. The first finds JBoss failing to correctly create advice, leading to the JVM message: “NoClassDefFoundError: .../ExporterBaseAdvisor.” The latter two involve permission errors that generate an IllegalArgumentException.

With two exceptions, each application was run using its entire test suite and the resulting data aggregated. The first exception was a unit test of JMeter that caused an error and was thus dropped. The second exception was GanttProject, which came with examples as its test cases. GanttPro-

<table>
<thead>
<tr>
<th>Program</th>
<th>LoC</th>
<th>Classes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GanttProject 2.0.9</td>
<td>69K</td>
<td>564</td>
<td>Project Scheduling</td>
</tr>
<tr>
<td>jasmin 2.3</td>
<td>40K</td>
<td>216</td>
<td>Java Assembler</td>
</tr>
<tr>
<td>jess</td>
<td>no src</td>
<td>460</td>
<td>Sandia Rule Engine</td>
</tr>
<tr>
<td>jmeter 2.3.4</td>
<td>147K</td>
<td>792</td>
<td>Testing Tool</td>
</tr>
<tr>
<td>jolden</td>
<td>6215</td>
<td>20</td>
<td>Olden Bench Mark</td>
</tr>
<tr>
<td>jtopas</td>
<td>24K</td>
<td>65</td>
<td>Java Tokenizer Lib</td>
</tr>
<tr>
<td>nanoxml 1.2</td>
<td>95K</td>
<td>611</td>
<td>Java Parser</td>
</tr>
<tr>
<td>siena 0.9</td>
<td>98K</td>
<td>34</td>
<td>Event Services</td>
</tr>
<tr>
<td>Total</td>
<td>479K</td>
<td>2762</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2. Idealized Distribution Examples

Figure 3. Subject Programs
ject was run twice: in the first execution the program was immediately exited after initialization and in the second the house example was loaded.

During execution, the tracer advice records each object instantiation as a chain of classes from the class created to java.lang.Object. In a post processing step, these chains are used to generate, for each class instantiated, the number of times this value occurs in the data. As formalized in Hypothesis 1, Research Question 1 represents a preliminary check that, for non-leaf classes in the dynamic inheritance hierarchy, multiple subclasses are instantiated. To address this question, the collected data was used to compute the number of subclass instantiations for each non-leaf class (except, as mentioned above, Object). The resulting data is shown in Figure 4 where the x-axis shows, for non-zero entries, the number of subclasses instantiated and the y-axis shows, using a log scale, the number of times this value occurs in the data.

The graph shows a considerable number of classes with more than one instantiated subclass (165 of 349, or 47%). One of these classes has 48 different instantiated subclasses. Statistically, the mean number of subclasses instantiations is different from 1.0 ($p = 0.00044$ for a one-sample t-test). Thus, there is evidence to reject the null hypothesis and conclude that more than one of a class’s subclasses is instantiated a substantial amount of the time.

The collected data also highlights the broad and shallow nature of the inheritance tree, which is consistent with the following trend: “From leading books in the field, you can observe that in the late ‘80s and early ‘90’s, inheritance was all the rage [but by] the mid-1990s, there was a major shift to composition rather than inheritance” [5]. With leaves accounting for 1718 of the 2120 instantiated classes (2120 is the dynamic class count and thus does not match the static count shown in column three of Figure 3), the following speculative optimization akin to copy on write may be worthy of investigation [10]. The optimization creates a specialized version of a class that assumes the class is final and injects a check to test if this assumption is ever violated. Overall performance gains occur if the gain from the specialized version outweighs the cost of the check.

4 Empirical Results

This section presents the empirical results used to address each of the three research questions. In doing so, the subclass instantiations of each class are considered, although Object is often discounted as it represents an artificial root-of-all classes rather than part of the problem decomposition.

A. Hypothesis 1

H0: only one of a class’s subclass is ever instantiated.

H1: more than one of a class’s subclasses are instantiated.

As formalized in Hypothesis 1, Research Question 1 represents a preliminary check that, for non-leaf classes in the dynamic inheritance hierarchy, multiple subclasses are instantiated. To address this question, the collected data was used to compute the number of subclass instantiations for each non-leaf class (except, as mentioned above, Object). The resulting data is shown in Figure 4 where the x-axis shows, for non-zero entries, the number of subclasses instantiated and the y-axis shows, using a log scale, the number of times this value occurs in the data. The graph shows a considerable number of classes with more than one instantiated subclass (165 of 349, or 47%). One of these classes has 48 different instantiated subclasses. Statistically, the mean number of subclasses instantiations is different from 1.0 ($p = 0.00044$ for a one-sample t-test). Thus, there is evidence to reject the null hypothesis and conclude that more than one of a class’s subclasses is instantiated a substantial amount of the time.

The graph shows a considerable number of classes with more than one instantiated subclass (165 of 349, or 47%). One of these classes has 48 different instantiated subclasses. Statistically, the mean number of subclasses instantiations is different from 1.0 ($p = 0.00044$ for a one-sample t-test). Thus, there is evidence to reject the null hypothesis and conclude that more than one of a class’s subclasses is instantiated a substantial amount of the time.

To address this question, the collected data was used to compute the number of subclass instantiations for each non-leaf class (except, as mentioned above, Object). The resulting data is shown in Figure 4 where the x-axis shows, for non-zero entries, the number of subclasses instantiated and the y-axis shows, using a log scale, the number of times this value occurs in the data. The graph shows a considerable number of classes with more than one instantiated subclass (165 of 349, or 47%). One of these classes has 48 different instantiated subclasses. Statistically, the mean number of subclasses instantiations is different from 1.0 ($p = 0.00044$ for a one-sample t-test). Thus, there is evidence to reject the null hypothesis and conclude that more than one of a class’s subclasses is instantiated a substantial amount of the time.

4.1 Hypothesis 2

H0: the distribution of subclass instantiations is uniform.

H1: the distribution of subclass instantiations is skewed.

To investigate Research Question 2, the percent of instantiations for each class are first ranked from highest to lowest. Only those classes with at least two instantiated subclasses were considered. The initial analysis considers the average for each rank; thus, the first average is the average of the subclasses having the largest percentage of the instantiations. Statistically, the averages show significant skew (χ² p < 0.0001), which is visually apparent in Figure 5. The best fit is a power-law model with percent = 203 r⁻¹.⁹ where r denotes the rank in the sorted order of percents as shown on the x-axis (1 · · · 48). The curve is a good fit (P, 0.773, –1.9). It explains 77.3% of the variation in the percentages. While not separated out in the figure, this same pattern is evident individually for each of the programs.

Thus, the aggregated data shows a clear skew. Turning to the individual classes statistically, χ² tests applied to each class find significant skew with 53% of the classes. Furthermore, 84% of these have a p value less than 0.0001. Thus, when a skew exists it is often dramatic.

Graphing the ranked data for each class can be used to visualize (as in Figure 2) the distributions of each class. The chart in the lower left of Figure 6 shows all 165 classes with at least two instantiated subclasses. The graph is only intended to provide an overall feel for the data. In this case it has a long tail that obscures much of the detail. The middle graph of Figure 6 shows only the first ten x-axis values. In this graph the appearance of horizontal lines is apparent.
Beginning with Hypothesis 3a, as a preliminary check 63% (104 of 165) of the classes find the class with the highest percentage of subclass instantiations having at least 50% of those instantiations. To formalize this initial indication of dominance, this section presents three linear models fit to subsets of the data. In contrast with the power-law model that fits the overall data rather well, the linear models are not good fits; however, they do provide two measures (slope and \( R^2 \)) of how close a distribution is to being uniform.

For a completely uniform distribution the slope is always zero and the \( R^2 \) value 1.00 (see Figure 2). In such a model, ignoring any one of the values will leave the same slope and \( R^2 \) value. Now consider the situation in which exactly one value is higher than the rest and thus the sole cause of non-uniformity. In this situation, the original linear model will have a negative slope, but after removing the one higher value, the slope will become zero and the \( R^2 \) value increase to 1.00. Thus a lower slope and a higher \( R^2 \) are both indicators of a more uniform distribution.

The comparison begins with all the data (the black line in Figure 5). A linear model for this data is then compared to linear models computed without the most frequently instantiated subclasses. The complete data set is described by a line having the slope \(-31\%\) and \( R^2 = 0.26 \). (Comparing this with the \( R^2 \) of 0.773 for the power-law fit shows how much more appropriate a power-law model is.) Removing the most frequently instantiated subclass, the slope falls to \(-19\%\) with an \( R^2 \) of 0.37. Ignoring the top two, it becomes even more uniform with a slope of \(-14\%\) and an \( R^2 \) of 0.55. The increasing uniformity (lowering slope and increasing \( R^2 \) value) when ignoring the most common classes combined with the finding that for 63% of the classes the most common subclass accounts for at least 50% of the instantiations, clearly support the alternative hypothesis that one or a small number of subclasses dominate the distribution.

Hypothesis 3b is a check on Hypothesis 3a. Visual evidence that such a check is needed can be seen in the center chart of Figure 6 where the long horizontal lines indicate classes with uniform distribution. To investigate the source of classes showing uniform distribution, classes were partitioned into two groups: those occurring frequently and those occurring infrequently. To produce a conservative cut-off, a class that accounted for 0.1% (one thousandth) of the total instantiation count was considered to have occurred frequently. The results are largely unaffected by the use of more standard cutoff values. For example, using outliers as determined from three times the inter quartile distance [17] to predict frequently instantiated classes produces a smaller frequent set and a stronger result than using 0.1%, and thus reinforces the results below.

Visually, the upper right graph in Figure 6 shows the distribution for the frequent group. Notice the absence of horizontal lines and thus classes with uniform distribution. To investigate the source of classes showing uniform distribution, classes were partitioned into two groups: those occurring frequently and those occurring infrequently. To produce a conservative cut-off, a class that accounted for 0.1% (one thousandth) of the total instantiation count was considered to have occurred frequently. The results are largely unaffected by the use of more standard cutoff values. For example, using outliers as determined from three times the inter quartile distance [17] to predict frequently instantiated classes produces a smaller frequent set and a stronger result than using 0.1%, and thus reinforces the results below.

Visually, the upper right graph in Figure 6 shows the distribution for the frequent group. Notice the absence of horizontal lines and thus classes with uniform distribution. To investigate the source of classes showing uniform distribution, classes were partitioned into two groups: those occurring frequently and those occurring infrequently. To produce a conservative cut-off, a class that accounted for 0.1% (one thousandth) of the total instantiation count was considered to have occurred frequently. The results are largely unaffected by the use of more standard cutoff values. For example, using outliers as determined from three times the inter quartile distance [17] to predict frequently instantiated classes produces a smaller frequent set and a stronger result than using 0.1%, and thus reinforces the results below.

C. Hypothesis 3

Research Question 3 further investigates the distribution of subclass instantiations. It has two parts formalized in the following two hypotheses.

Hypothesis 3a

H0: most subclasses take part in the skew.
Ha: one or a small number of subclasses dominate.

Hypothesis 3b

H0: skew is not correlated to instantiation count.
Ha: higher instantiation count correlates to greater skew.
Quantitatively, repeating the second analysis used with Hypothesis 3a for the frequent group, a linear model has a slope of $-27\%$ with $R^2 = 0.16$. By ignoring the top most instantiated class this falls to $-12\%$ with $R^2 = 0.26$, and ignoring the top two it further falls to $-7\%$ with $R^2 = 0.48$. Two observations can be made. First, the $R^2$ values are lower in each case, indicating a “less-linear”, and thus less uniform, fit. The second observation comes from comparing the change in slope after ignoring the top most instantiated class. For all the data, the change is less (31% to 19%) than with the top 99.9% from the frequent group where it falls from 27% to 12%. Here again the statistics indicate that uniformity is coming in the infrequent group, which represents a very small fraction of the data.

Finally, from Hypothesis 2 the $\chi^2$ test found that 53% of all classes showed significant skew. Only one of these, CompositeTag, is from the infrequent group. This class is considered below in the case study. Thus the frequent group is dominated by classes with significant skew. Further evidence that uniformity is primarily found in lightly instantiated classes comes from considering the percentage of instantiations attributed to classes showing significant skew: such classes account for 99.98% of the instantiations.

The data above supports the alternative hypotheses for both Hypothesis 3a and 3b. This result suggests that a class instantiation tracking system need only track and attempt to manipulate the top handful of instantiated classes. Doing so saves the cost of more complete tracking.

4.4 Case Studies

This section considers the eight individual programs in more detail. The programs are divided into three groups: the first, nanoxml, siena, and jolden make little use of inheritance. The second group of mild inheritance users includes jasmin, jtopas, and jess, with GanttProject and jmeter making up the final group of heavy inheritance users.

Figure 7 shows the inheritance hierarchies for all eight programs, with a focus on one from each group (jolden, jess, and jmeter)\(^2\), taken as the case studies. Note that Figure 7 includes considerable detail that is not readable without zoom capability. This detail is retained in the figure to give an impression of each chart and for the benefit of online readers. Rather than showing classes of each hierarchy as simple rectangles (as done, for example, in the UML), each class is represented as a pie chart indicating the distribution of subclass instantiations. For example, class Node of jolden (shown in the upper left of the figure) has two subclasses: Cell, instantiated 82.77% of the time, and Body, instantiated the other 17.23% of the time.

Because their distributions are uninteresting, classes with zero or one instantiated subclass are grouped together in their superclass. This also serves to de-clutter the presentation of the hierarchies. In all, five classes were effected. For example, the pie chart for Object of jolden shows three regions: 0.36% of the instantiations are Nodes, 9.44% are QuadTreeNode and the remaining 90.20% are from classes with zero or one subclass (and thus have no interesting distribution). Since jolden is a computational benchmark and a simpler program, it is dominated by this set.

The case study subject taken from the first group is the computational benchmark jolden, which has two classes that themselves have subclasses: Node and QuadTreeNode (which, despite its name, is not a subclass of Node). Both of these subclasses show significant skew in their subclasses instantiations. For Node’s two subclasses, almost five times as many instantiations of class Cell are created compared to instantiations of class Body. Class QuadTreeNode has three instantiated subclasses showing a breakdown of 38%, 37%, and 25%. This is actually one of the more uniform distributions seen outside some special cases discussed below. It is interesting to note that despite it being an artificial root-of-all superclass, the distribution of the instantiations for Object’s 36 subclasses show an exponential decay similar to the overall data (E, 0.96, $-0.39$).

Next, siena has the flattest class hierarchy of all eight programs. The only classes that do not have Object as their superclass deal with exceptions and are instantiated only 0.08% of the time. It is again interesting to note that the instantiations of Object’s subclasses have a similar shape to the overall data (E, 0.95, $-0.32$). Finally, nanoxml also has a flat, almost non-existent, class hierarchy except for two kinds of readers and three database elements. The readers are highly skewed with over 96% being ContentReaders. The database elements show less skew with subclasses having 49%, 26%, and 25% of the instantiations. As with the other two programs, the 14 subclasses of Object show a skewed distribution (P, 0.97, $-1.50$). In this case, the power-law distribution is more extreme than that of the other two.

Moving to the second group, 11 of jtopas’s 15 classes have a single subclass instantiated. The four remaining classes (excluding Object) each show significant skew ranging from (P, 0.86, $-0.79$) to (P, 0.93, $-1.10$). Finally, the subclasses of Object again show a skewed distribution (E, 0.92, $-1.19$). Also in this group, jasmin has six classes (excluding Object) that themselves have subclasses. Those having more than one instantiated subclass all show skewed distributions ranging from (P, 0.80, $-2.20$) to (P, 0.98, $-1.53$). The comparatively low $R^2$ value of the first comes from the uniform tail in its distribution: 90.5%, 2.4%, 2.4%, 2.4%, 1.2%, 1.2%. As explained below in a similar example, this occurs when two distributions (one skewed and one uniform) are overlaid. Finally, Object again shows a strong power-law skew (P, 0.96, $-3.17$).

For the second group, jess, shown larger in Figure 7, was chosen for a more in depth consideration because it includes patterns seen in jasmin and jtopas and also includes classes with uniform distributions. Ten of jess’s 303 instantiated classes have a single instantiated subclass and 282 have none. Of the remaining eleven classes (twelve including object), six have their subclasses instantiated exactly 14 times and one (part of the GUI) has its two sub-

---

\(^2\)Full size versions of each distribution hierarchy can be found at [http://www.cs.loyola.edu/~binkley/subclass-distributions](http://www.cs.loyola.edu/~binkley/subclass-distributions)
Figure 7. Example Class Hierarchies. The figure is intended to convey an overall feeling for the distributions. It is not intended to be completely readable. Full size versions of all eight distributions can be found at http://www.cs.loyola.edu/~binkley/subclass-distributions.
classes instantiated exactly once each. This produces the uniform distributions evident in Figure 7 by the symmetric pie charts (see, for example, jess.e1 shown within a dashed box). Three classes have a frequency of 28 rather than 14: jess.fa, jess.d9, and jess.du. The latter of these three accounts for the larger wedge in the lower right pie chart for jess.c). An explanation for these distributions is found in an inspection of the test suite, which includes 14 test cases (two of which make use of the GUI). This pattern is repeated in many of the GUI examples where the dominant source of uniform distributions is GUI elements instantiated once per test.

The remaining four classes (not counting Object) are visible in Figure 7 as having non-uniform distributions. The one with only two instantiated subclasses shows a mild skew, with the more common subclass accounting for 76% of the instantiations. The remaining three all show significant skewed distributions (E, 0.97, −0.50), (E, 0.86, −0.58), and (P, 0.998, −2.00). Finally, as seen before, Object shows a strong skew (P, 0.97, −1.60).

In the final group, GanttProject’s 522 instantiated classes include 408 with no subclass, 69 with a single instantiated subclass, 23 with two instantiated subclasses, and 22 with three or more instantiated subclasses. Of those with two subclasses, only one is instantiated a significant number of times. It has a uniform distribution split between RangeSearchFromKey and RangeSearchToKey. Such paired classes are the second cause of uniform distribution, the first being the GUI. The remaining classes showing a uniform distribution (e.g., e1.TestGanttRolleroverButton) are all associated with the GUI. From the data, a uniform distribution again appears to be an indicator of infrequent instantiation.

All but four of the 22 classes with 3 or more subclasses are instantiated very few times (less than 1% of the time). In contrast, with one exception, those instantiated a significant number of times have the expected skew. The exception, AbstractAction, is dominated by the subclass GPAction (accounting for 90% of the instantiations) but then has a long flat tail. It thus shows evidence of two overlaid distributions. It thus shows significant skew, but also shows features of a uniform distribution. This overlay of two distributions suggests that AbstractAction finds two uses in the program.

The final system, jmeter, produces the largest inheritance hierarchy. As seen in Figure 7, it includes features seen in the smaller hierarchies. Of the 1043 classes instantiated, 869 have no instantiated subclass, 85 have a single instantiated subclass, and 38 have exactly two instantiated subclasses. This leaves 51 classes with three or more instantiated subclasses. Here again, uniform distributions most often accompany classes related to the GUI. Eleven classes have completely uniform distributions. Seven of these represent elements of the GUI, two support the Bean Scripting Framework, and two the parsing of HTML input. One example, class CompositeTag, has 27 subclasses, each instantiated 18 times. These come from 18 test cases where jmeter builds an HTML parser that includes support for the 27 HTML tag types that are subclasses of CompositeTag.

The remaining classes show a skewed distribution or a hybrid of the skewed and uniform distributions. An example hybrid is the class AbstractSamplerGui, whose distribution includes a 46% followed by 3.6% repeated 15 times. This is accounted for when looking at test cases where the 46% comes from the heavily tested subclass HttpSessionSamplerGui, while the rest only get mentioned in coverage tests.

The rest of jmeter’s instantiated classes show a skewed distribution except AbstractList, which is primarily split between ContentList and AttributeList (49.64% and 49.61% respectively). Two other subclasses split the remaining 0.72% of the instantiations. This example along with jodd’s QuadTreeNode make up the only two classes showing such a split for a non-trivial non-GUI class.

4.5 Threats to Validity

There are three threats relevant to this research: statistical conclusion validity, external validity, and internal validity. The fourth, construct validity, does not represent a serious threat as all of the variables used in the study accurately measure the concepts they claim to measure.

The only serious threat to statistical conclusion validity arises from some of the instantiation counts being quite large for use with the $\chi^2$ test. In all cases multiple statistical tests were employed to help reinforce the findings.

External validity, sometimes referred to as selection validity, is the degree to which the findings can be generalized to other (external) settings. The experiment’s primary external threat arises from the possibility that the selected programs are not representative of Java programs in specific and object-oriented programs in general. This is a reasonable concern that applies to any study of program properties. The selected set is clearly not random as it represents a convenience sample (due to the requirements that the selected programs include a sufficient test suite and be compatible with JBoss). To mitigate this concern the study considers a large code base chosen for its diversity. For example, it includes GUI and non-GUI programs, a computational benchmark, and programs from a range of application areas including end-user applications, utilities, and static analysis tools. There is, therefore, reasonable cause for confidence in the results obtained and the conclusions drawn from them.

Finally, potential threats to internal validity, for example, history effects, attrition, and subject maturation are non-issues given the study’s short duration and lack of human subjects. The only internal threat comes from the test suite. This threat includes the confounding effect of the test suite when uniform distributions are seen because a class is instantiated once per test case. In addition, the provided test suites, which were used to avoid experimenter bias, may not represent typical use or provide 100% statement coverage.

5 Technique Informing

A better understanding of subclass instantiation distribution can be used to inform existing techniques. Several examples were given in the introduction. In addition to these examples, this section takes a more detailed look at
one representative technique from those that stand to benefit from a better understanding of subclass instantiation distribution. The representative example is the object-oriented cohesion metric work of Al Dalla and Briand [6]. Such metrics form an important part of assessing object-oriented software quality. With this in mind, Al Dalla and Briand introduce and formalize a cohesion metric based on the similarity between pairs of methods and pairs of attribute types in a class. For example, attribute-attribute cohesion arises when a pair of attributes is accessed by a common method. The metric is validated both theoretically, by showing that it satisfies a collection of mathematical properties, and empirically, using the study of four open source systems.

A better understanding of subclass instantiation distribution impacts their technique in two ways. First, knowing the distribution can lead to efficiency improvements, which is of particular importance to a cohesion metric because, in practice, assessing cohesion must be done frequently and automatically. The goal of the second impact is to improve the precision of the technique. For example, consider the cohesion metric presented with two pairs of attributes. Knowing that the distribution is not uniform means that treating all attributes the same is safe, but the least accurate approach. By incorporating subclass instantiation information, methods from subclasses instantiated more often can provide greater cohesive force.

6 Related Work

Several projects have found power laws in static metrics associated with Java programs [20, 2, 19]. DuFour et al. consider a wide range of dynamic metrics ranging from simple size counts to measures of concurrency [7]. Several of these summarize the data presented herein. For example, the appReceiverArity metric captures the percentage of all call sites that have one, two, or more different receiver types.

The remainder of this section samples two analyses that both have a goal similar to the distribution analysis. The first analysis, performed by Moret, Binder, and Villalón [16], concerns calling-context profiling, a technique for locating hotspots in programs by separately considering each dynamic calling context. The technique can be used to produce metrics such as the CPU time spent in a given calling context. Instrumenting Java bytecode using a technique similar to, but more sophisticated than, the weaving used to collect the object instantiation data, each Java method is transformed to maintain a Calling Context Tree as an additional method parameter. The instrumentation encounters several issues similar to those encountered using JBoss. For example, Moret, Binder, and Villalón describe how in Sun’s JDKs there are certain methods that rely on a fixed invocation sequence, which is disrupted by the introduction of advice. Examples occur in java.lang.Class, java.lang.ClassLoader, and java.lang.Runtime.

The second analysis by Yang, Dwyer, and Rothermel considers a new technique for regression model checking [21]. While effective at verifying program behavior, model checking can be too expensive to exhaustively apply after each modification. The new technique applies model checking incrementally, which is shown to be significantly faster. An important part of the analysis is the computation of a reachable elements map, which includes dynamically reachable elements (e.g., basic blocks). This information is obtained by instrumenting a program’s bytecode using BCEL. A tool similar to JBoss.

These two tools see a potential improvement from being better informed about the distributions of subclass instantiations. For example, in the model-checking case, knowing that subclass instantiations are skewed and furthermore having access to past program executions, as is common during regression testing, allows information about subclass distribution to improve the incremental model checking. One such improvement involves the computation of the reachable elements map which might include a simple strategy of caching the most common subclass of each class. A more complex approach could exploit the lack of uniformity similar to the profiling technique of Ball and Larus [1]. Their approach did not instrument the most common path in a control-flow graph, but instead computed the profile information for this path using its parent path and the information for its less common siblings. In a similar fashion, a model checker might not instrument the most common subclass, but instead derive the information required for that class from that of the other subclasses and its superclass. This provides ever greater performance improvement as the non-uniformity of the distribution grows more extreme.

7 Future Work

This section considers two avenues for future investigation. To begin with, future work will look for correlations with static code metrics (e.g., size, cohesion, attribute count) [9, 14, 15]. Finding static properties that predict the distribution would allow an engineer to focus effort even before dynamic information was available.

Another avenue of future work considers the relation between the distributions of system, GUI, and user classes. For example, consider the hypothesis that user code makes less use of inheritance. A potential motivation for such a hypothesis follows from the assumption that system and GUI code is more carefully laid out and slower to evolve. In addition, GUI code typically lends itself to deeply nested class hierarchies. Thus this code is different from user code where a deeply-nested class hierarchy exacerbates the cost of frequent requirement changes and imminent deadlines.

A preliminary test of this hypothesis was performed by first tagging each class as system, GUI, or user, based on its name. After removing Object, each chain of class instantiations was then broken down into three (possibly empty) parts: system classes, GUI classes, and user classes.

The average length for each chain part is shown in Figure 8. Using the data from all 2430 chains, preliminary statistical analysis (using Dunn’s procedure) finds two groups with system and user chains being on average shorter than GUI chains. As the original data was not gathered with this experiment in mind, this result is a preliminary, but still an interesting, start.
<table>
<thead>
<tr>
<th>Program</th>
<th>All</th>
<th>System</th>
<th>GUI</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>GanttProject</td>
<td>2.01</td>
<td>1.43</td>
<td>2.16</td>
<td>1.33</td>
</tr>
<tr>
<td>jasmin</td>
<td>1.70</td>
<td>-</td>
<td>-</td>
<td>1.70</td>
</tr>
<tr>
<td>jess</td>
<td>1.25</td>
<td>1.00</td>
<td>3.00</td>
<td>1.25</td>
</tr>
<tr>
<td>jmeter</td>
<td>2.66</td>
<td>1.57</td>
<td>4.11</td>
<td>1.99</td>
</tr>
<tr>
<td>jolden</td>
<td>1.14</td>
<td>-</td>
<td>-</td>
<td>1.14</td>
</tr>
<tr>
<td>jtopas</td>
<td>2.40</td>
<td>4.25</td>
<td>-</td>
<td>1.69</td>
</tr>
<tr>
<td>nanoxml</td>
<td>1.55</td>
<td>1.50</td>
<td>-</td>
<td>1.25</td>
</tr>
<tr>
<td>siena</td>
<td>1.11</td>
<td>2.00</td>
<td>-</td>
<td>1.00</td>
</tr>
<tr>
<td>Project Average</td>
<td>1.73</td>
<td>1.96</td>
<td>3.09</td>
<td>1.42</td>
</tr>
<tr>
<td>Chain Average</td>
<td>2.21</td>
<td>1.71</td>
<td>3.03</td>
<td>1.58</td>
</tr>
</tbody>
</table>

Figure 8. Average dynamic inheritance tree heights for system, GUI, and user chains. A “-” denotes an unseen chain part excluded from the average. Recall that JBoss cannot weave into system-class constructors and thus the data only samples the creating of system classes by including those whose chains begin with a user or library class. Project average is the average of each program (column) while Chain average is the average of all chains, which means programs producing more chains have a greater influence.

8 Summary

This paper considers the instantiation distributions of Java classes. With a few exceptions, such as GUI classes, these distributions are heavily skewed to the right. This means that a few classes (often one class) are the dominant subclasses instantiated. Knowing that such distributions are heavily skewed allows programmers, testers, managers, maintainers, tool designers, etc. to better focus their efforts.

9 Acknowledgements

Support for this work was provided by Loyola’s Hauber Summer Research Program and NSF grant CCF 0916081. Thanks to Dawn Lawrie for her insightful comments, to Chris Morrell for his help with the statistical analysis, and to Jennifer Ellis, Matt Hearn, Stephanie Lukin, and Austin Wheeler for their comments on earlier drafts. In addition the anonymous reviewers provided extremely thoughtful reviews. For this we are very grateful.

References