How Do Java Programs Use Inheritance? An Empirical Study of Inheritance in Java Software

Ewan Tempero[1], James Noble[2], and Hayden Melton[1]

[1] Department of Computer Science, University of Auckland, Auckland, New Zealand
ewan,hayden@cs.auckland.ac.nz
[2] School of Mathematics, Statistics, and Computer Science, Victoria University of Wellington, Wellington, New Zealand
kjx@mcs.vuw.ac.nz

Abstract. Inheritance is a crucial part of object-oriented programming, but its use in practice, and the resulting large-scale inheritance structures in programs, remain poorly understood. Previous studies of inheritance have been relatively small and have generally not considered issues such as Java's distinction between classes and interfaces, nor have they considered the use of external libraries.

In this paper we present the first substantial empirical study of the large-scale use of inheritance in a contemporary OO programming language. We present a suite of structured metrics for quantifying inheritance in Java programs. We present the results of performing a corpus analysis using those metrics to over 90 applications consisting of over 100,000 separate classes and interfaces. Our analysis finds higher use of inheritance than anticipated, variation in the use of inheritance between interfaces and classes, and differences between inheritance within application types compared with inheritance from external libraries.

1 Introduction

Since the introduction of the object-oriented paradigm, much has been written on the notion of "inheritance" [1]. To some, the very idea of "object-orientedness" is bound up in inheritance [2,3]. Inheritance does appear to be very prominent in discussions about good design. All the design patterns have it [4], frameworks depend on it [5] and it's even in UML [6].

Some presentations of the object-oriented paradigm (in textbooks for example) place so much importance on inheritance that the implication is that any design without "lots of inheritance" is not a good one (or certainly not "object-oriented"). At the same time, there is a considerable amount of advice urging caution with respect to use of inheritance, such as "Favor object composition over class inheritance" [4]. There have also been studies providing conflicting answers as to its benefits [7,8,9], but also suggesting that "too much" inheritance is detrimental.

The aim of this paper is to answer a simple question: ***"How do programs use inheritance?"***. To make this question concrete, we address this question to Java, thus: "How do Java™ programs use inheritance?".

To answer this question, we first consider how Java supports inheritance. Compared with earlier object-oriented languages such as Smalltalk, Eiffel, or C++, Java distinguishes between *extends* and *implements* relationships. To understand how Java (and

J. Vitek (Ed.): ECOOP 2008, LNCS 5142, pp. 667–691, 2008.
© Springer-Verlag Berlin Heidelberg 2008

other languages making this distinction, such as C♯) actually use inheritance, we need to consider each relationship individually. We also consider other issues regarding inheritance — for example, we treat inheritance from one of the library classes as being different to inheritance from another class defined for the system. Our work is grounded in a systematic consideration of all these issues, resulting in a structured suite of metrics for measuring the various kinds and usages of inheritance in Java programs.

The suite of inheritance metrics we propose provides a sensitive instrument for characterising various types of inheritance in a particular program. To give an overall answer to our question — how do Java programs *in general* use inheritance — we gathered a substantial corpus of 93 open-source Java applications, including over 100,000 user-defined types. Then we applied our metrics to this corpus, with the resulting distribution of metrics values characterising the use of inheritance in that corpus, and hopefully getting as close to ***accepted practice*** in Java programs as possible.

A key point about the methodology we use here is that it is primarily *descriptive*: our research question asks simply "what do Java programs do?" We are interested in understanding "Java as she is spoke" — that is, in the way Java programs are actually structured in the real world — rather than how we fondly imagine Java programs should be written. As our terminology suggests, we draw on the established methodology of corpus linguistics. Our corpus is collected from large, well-known, widely-used Java programs (such as Eclipse, Open Office, Spring, Tomcat) — programs that are apparently well regarded by other Java programmers, and that we believe constitute as much a representative sample of Java programs "in the wild" as any other.

In evaluating individual programs (from the corpus or outside it) we can discuss whether their use of inheritance is *typical* or *extreme* with respect to the corpus, that is whether their use of inheritance seems relatively close to accepted practice embodied by the corpus, or whether and how it diverges. This is not to say we are uninterested in questions of how inheritance could or should be used in the abstract — just that those questions are separate from the questions about how inheritance is actually used in accepted Java practice, and we do not address them in this paper.

The paper makes the following contributions:

- A fine grained, structured suite of inheritance metrics for Java-like languages.
- A corpus analysis applying these metrics to 93 Java applications containing over 100,000 user-defined types.

Based on the corpus analysis, we demonstrate some important features of the accepted practice regarding inheritance in Java programs:

- most classes in Java programs are defined using inheritance from other "user-defined" types.
- classes and interfaces are used in stereotypically different ways, with approximately one interface being declared for every ten classes.
- client metrics have truncated curve distributions while supplier metrics have power law-like distributions.
- most types (classes and interfaces) are relatively shallow in the inheritance hierarchy.
- almost all types have fewer than two types inheriting from them: however for some very popular types, the bigger the programs, the more types will inherit from them.

– larger (or older) systems make proportionally more use of inheritance from user-defined classes, and less use of standard library or third-party library classes.

The rest of this paper is organised as follows. In the next section, we summarise the related work. Section 3 discusses various issues regarding the characterisation of inheritance that need to be considered when measuring it. Section 4 presents the metrics we used in our study; our results from collecting these metrics are presented in section 5. Section 6 presents a discussion of our results, and finally we give our conclusions and discuss future work in section 7.

2 Background and Related Work

The most often mentioned inheritance related metrics are Chidamber and Kemerer's DIT and NOC metrics [10,11]. DIT for a class is defined as the length of the longest path from the class to the root of the inheritance hierarchy it is in. The authors argue that the deeper the class, the more complex it would be as it would inherit from more ancestors, but also the more potential reuse there could be. NOC for a class is defined as the number of immediate subclasses of that class. The authors suggested that more children means more reuse, but also the greater the likelihood of improper abstraction. They also observed that NOC gives an idea of the influence a class has on the design.

DIT and NOC were introduced in 1991 but it was not until the 1994 publication that Chidamber and Kemerer presented measurements using them. The measurements were of two sites. One site consisted of two graphical user interface libraries with 634 C++ classes. The other consisted of class libraries used in the implementation of a computer aided manufacturing system for the production of VLSI circuits and had 1459 Smalltalk classes.

For DIT, the C++ site had a median value of 1 and maximum of 8, whereas the Smalltalk site had a median of 3 and maximum of 10. However, it was noted that for Smalltalk, all classes are subclasses of the class "Object", meaning that only "Object" could have a DIT measurement of 0. For NOC, 73% of the C++ classes and 68% of the Smalltalk classes had no children. The maximum NOC measurements reported were 42 for C++ and 50 for Smalltalk. It is worth noting that the results in this paper were presented as a frequency distribution. This presentation means that we can determine such things as for the C++ site about 200 classes had a DIT of 0 (meaning about 400 classes had a non-zero DIT) and just under 300 classes in the Smalltalk site had a DIT of 1 (and so 1100-1200 classes had a DIT of more than 1).

There have been various efforts to establish the veracity of Chidamber and Kemerer's thinking or similar claims about inheritance. We report only the most relevant to our work.

Daly et al. carried out an investigation on the impact of depth of inheritance on maintenance as measured by the amount of time taken to perform a maintenance task [7]. Their results suggested that inheritance had a negative effect on maintenance time. This study was later replicated, with the results suggesting the opposite effect — that inheritance had a positive effect on maintenance time [8]. That these two studies could get such different results suggests there may be more to inheritance than just "depth,"

although both were sufficiently small that it is possible that some effect other than inheritance was observed.

Another replication was carried out by Harrison et al. [9]. They studied two C++ systems, each with two versions. One system had a version without inheritance consisting of 360 LOC and a version with 290 LOC with maximum DIT of 3. The other system had one version with 1200 LOC and the other with 900 LOC and maximum DIT of 5. Their results suggest that inheritance made it harder to modify systems, but that size and functionality of a system may affect understandability more than the "amount of inheritance" used. The authors observed that an external threat to the validity of their results was the small size of their system. They claimed that the levels of inheritance investigated were "typical of those found in larger systems." Our interest is in whether DIT is sufficient to characterise "amount of inheritance", and whether the systems used in this and earlier studies really can be considered "typical."

On the question of "how inheritance is used", there appears to be little in the way of published results. Manel et al. [12] try to determine differences in maintainability of code written in the OO paradigm vs the structured programming paradigm, which included use of metrics for inheritance. The two inheritance metrics they use are "number of derived classes", and the number of lines of code in the classes a derived class inherits from (in order to try and gauge the degree of reuse via inheritance). They look at 5 versions of one "medium sized" application from the telecommunications domain. The first version had 20 derived classes out of 57, while the 5th version had 87 derived classes out of 225 (39%).

One of the largest studies that we are aware of is by Succi et al. [13]. They applied the metrics suite by Chidamber and Kemerer to 100 Java and 100 C++ applications. They were investigating the statistical properties of the CK metrics but our interest is in the metric values they report. The Java applications ranged from 28 to 936 classes in size (median 83.5) and the C++ applications ranged from 30 to 2520 classes (median 59). The actual applications were not identified. Interpreting their box plots, the DIT measurements for the Java applications were mostly in the range 2-5, with outliers at 10. For the C++ applications, most measurements were in the range 5-6 with outliers both above and below, and a maximum of 9. For NOC, the Java measurements were almost entirely less than 10, although there were outliers larger than 150, and for the C++ measurements, the range was similar, although there appear to be more outliers.

In another large study, Collberg et al. analysed 1132 Java jar files collected from the Internet [14]. According to their statistics they analyse a total of 102,688 classes and 12,188 interfaces. While no information was given as to what applications were analysed, this paper is good for the amount of data it provides. Much of it is not directly relevant to our study, but they did produce histograms of "inheritance graph height per application" finding a maximum of 10, a median of 4 and a minimum of 1, and also "number of user-class extenders per application" finding a maximum of 641, a median of 5, and a minimum of 0.

Other studies involving measuring DIT and NOC have judged the measurements to be "low", but consisted of quite small samples. Chidamber et al. studied three systems, one with 45 C++ classes, one with 27 Objective C classes, and one identifying 25 classes

in design documents [15]. The largest values they report are DIT of 3 and NOC of 11 (both from the design documents).

Basili et al. investigated the Chidamber and Kemerer metric suite as predictors of fault-prone classes [16]. Their study consisted of 8 teams of 3 students building a "medium-sized" video rental system, resulting in 8 C++ applications consisting of 180 classes in total. The maximums they observed were 4 for DIT and 5 for NOC. However it should be noted that with 180 classes spread over 8 applications, each application must be fairly small.

Briand et al. carried out a study to determine to what extent various metrics are useful for detecting the probability of detecting faulty classes using the same applications as that by Basili et al. [17]. In addition to DIT and NOC, they also considered, number of parents (NOP), number of descendants (NOD), and number of ancestors (NOA). The maximums were 1 for NOP, 9 for NOD, and 4 for NOA.

Having a measurement by itself is of limited use. We also need what Kitchenham et al. refer to as the *entity population model*, which identifies the "normal values" of what is being measured under specific conditions [18]. They observe that without knowledge of such models, we cannot interpret what a measurement means. A starting point to developing such models for a metric is to apply it to "real" code, that is, code written as part of a software application, rather than code written to demonstrate the metrics, and, most importantly, *report the results*.

3 Characterising Inheritance

The starting point for our work was to determine a meaningful answer to the question "How much inheritance is being used in this application?" There are several reasons why having an answer to this question would be useful. This question is fundamental to evaluate any claims made about its benefits. If we cannot reliably measure inheritance use, then we cannot be sure that any changes that observed are due to (or just due to) inheritance. For example, in the studies on the effect of inheritance on maintenance described in the previous section, inheritance was characterised by just one measurement (DIT), whereas other inheritance metrics have been proposed (e.g. NOC).

Given the advice against overuse of inheritance, it would be useful to know to what extent this advice has been followed. We have previously seen a case where reality does not match theory [19]. Such situations may indicate problems with the theory, problems with its application (e.g., lack of appropriate tool support), or perhaps problems with training. We cannot determine which without appropriate metrics, and knowledge of how to interpret the measurements produced by the metrics.

To illustrate these points, consider the report by Chidamber and Kemerer on the Smalltalk site. They reported it consisted of about 1450 classes and had a maximum DIT of 10. How should we interpret this value of "10"? Chidamber and Kemerer characterised it as "rather small." But as we observed, other studies suggest that systems with measurements of 10 would have maintainability issues, implying it should be rare, which is what the study by Succi et al. suggests.

Another view of the meaning of a DIT value can be gained by considering a complete binary tree with the maximum number of nodes possible while also having a maximum

DIT of 10. Such a hierarchy would have $2^{11} - 1 = 2047$ nodes in it, which is comfortably more than the 1450 classes of the Smalltalk application. While we should not expect a realistic inheritance hierarchy to look like a binary tree, it does give an indication as to what the hierarchy must look like in order to get DIT measurements greater than 10. Its shape would have to somewhat more "tall and skinny". Our key observation is, we do not know what maximum DIT value we should *expect* for an application with 1450 classes.

In the complete binary tree hierarchy, the maximum NOC would be 2, and all but one class would inherit from some other class. Yet many variations on this are obviously possible: fewer classes could inherit from other classes while still having a maximum DIT of 10 and NOC of 2, the maximum DIT could be much greater than 10 while the NOC is no more than 2, the NOC can be much greater than 2 while the DIT is less than 10. On this basis we argue that it is not sufficient to characterise the inheritance hierarchy of an application by just one (or two) metrics.

Chidamber and Kemerer observed that the Smalltalk distribution is somewhat "top heavy", with the frequencies for DIT measurements 1-3 being around 300, and those for 4 and 5 being around 200. The Smalltalk distribution could be explained by the presence of the class library that is standard for any Smalltalk distribution. We speculate that many DIT distributions for Smalltalk software would be dominated by the library classes, meaning all distributions would look very similar. This raises the question as to how such standard libraries should be handled when defining metrics.

As we noted, Chidamber and Kemerer originally provided distributions of these metrics, and from these we can answer such questions as "how many classes have DIT or NOC of 0". The proportion of classes with DIT of 0 is the proportion of classes that do not use inheritance in their definition. Knowing whether this proportion is 70% rather than 90% would seem to give us some reasonable idea of the degree to which the application uses inheritance to define classes. Similarly it would be useful to know the proportion of classes with NOC of 0, that is, not providing inheritance relationships with other classes. Rather than rely on determining these values from frequency distributions, we will define such metrics directly.

Classes can be involved in inheritance in a number of different ways. For example, in Smalltalk, all classes except "Object" can be said to use inheritance. Perhaps we should distinguish those that inherit from "Object" from those that inherit from other classes. Taking this further, perhaps we should distinguish classes that inherit from any standard library class from those that are defined in the application. Those that inherit from a library class could be considered to be benefiting the most from reuse (since the library class doesn't have to be written) albeit restricted to what the library provides. A user-defined class inheriting from another user-defined class benefits less from reuse, but the designs of both classes are under the control of the software developer, and so this relationship could better represent the "quality" of the design.

The issue with measuring some kind of "DIT" metric in Java is the distinction between `extends` and `implements`. This distinction allows for a certain kind of "multiple" inheritance. The issue of measuring DIT in the context of multiple inheritance was defined by Chidamber and Kemerer to be the length of the longest path to the root [11]. However, if we are measuring a class that extends another class and implements

an interface, it's not clear that paths that follow the extends relationship are the same as those that follow the implements relationship. Rather than make a judgement, we will consider all variations.

Another issue in measuring DIT in Java is dealing with the situation where a type defined in the application inherits from a type for which we do not have the definition. For example, we create a new class `MyVector` that extends `java.util.Vector`. Since `Vector` has 3 ancestors, DIT (`MyVector`) should be 4. If we did not know `Vector`'s ancestry (or at least its DIT value), then we would not be able to measure DIT for `MyVector`. It is a limitation of our study that the corpus we use does not have the complete external libraries, making it impossible to determine the "true" DIT values in all cases. We will note that as far as the developer is concerned `Vector` (and any other type that is not part of the application) can usually be treated as if it were a "flattened" type. In the case of our example, we know `MyVector` extends `Vector`, but in terms of reasoning about `MyVector` it is not so relevant what `Vector`'s true structure is. For this reason we will define a variant of DIT (and similar metrics) that considers only the user-defined inheritance structure.

4 Inheritance Metrics for Java

4.1 Modelling Inheritance

In order to unambiguously define metrics for inheritance in Java we need to specify a model of Java inheritance. What we have been colloquially calling the inheritance "hierarchy" is really a directed acyclic graph (DAG), where the vertices are Java reference types and edges are inheritance relationships. For the purposes of the definition we always assume that `java.lang.Object` is in the graph, and that any classes without explicit superclasses have an edge to `Object`. To do otherwise would mean that some metrics change values if a programmer explicitly includes `extends Object`.

There are two kinds of edges, one for `extends` and one for `implements`. If `A extends B` then `A` is the *child* and `B` is the *parent*, similarly if `A implements B`. When we "follow an edge", we traverse the edge from child to parent.

The Java Language rules mean that at most one type of edge can connect any pair of vertices, and in some cases both may be disallowed. For example, an `implements` edge cannot occur between two class vertices. Practically speaking, the most common connections will be between pairs of classes, pairs of interfaces, or class-interface pairs. All others are very unlikely (e.g. enum implements annotation) or at least very rare (e.g. class implements annotation). Table 1 shows all the possibilities.

We have different kinds of vertices to distinguish different kinds of types, that is, *classes* (C), *interfaces* (I), *enums* (E), *annotations* (A) and *exceptions* (Ex). We distinguish classes and interfaces as they have quite different inheritance relationships with each other and play different roles in an inheritance hierarchy. We distinguish enums and annotations because, although they are respectively implemented as specialised classes and interfaces, their roles are somewhat different. Furthermore, they are in fact implemented in terms of inheritance (extending `java.lang.Enum` and `java.lang.annotation.Annotation` respectively), something that is evident at the bytecode level, although for the purposes of our metrics we will ignore these

Table 1. Allowable type inheritance relationships

	Class	Interface	Enum	Annotation	Exception
Class	extends	implements		implements	
Interface		extends		extends	
Enum		implements		implements	
Annotation					
Exception		implements		implements	extends

relationships. Finally, exceptions are distinguished from classes as they also have specialised roles, and furthermore are explicitly defined in terms of inheritance (extending `java.lang.Exception`). Combining exceptions with other classes when trying to determine the amount of inheritance may therefore give misleading results.

As well as ignoring the implicit inheritance relationships with `Annotation` and `Enum` we also ignore inheritance relationships with marker interfaces, specifically `java.io.Serializable` and `java.lang.Cloneable`.

For each kind of type there are 3 different kinds of vertices: user-defined (that come from the application we are measuring) standard library (from the Java Standard API), and third party (any remaining types from neither user code nor the standard library).

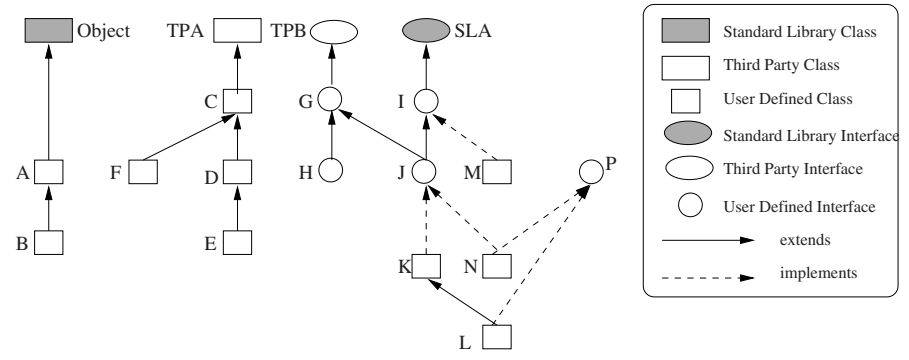
Each vertex has a "nesting level" attribute that indicates the level of nesting of the type represented by the vertex, where 0 indicates a top-level type, and the nesting level of any nested type (e.g., inner class or interface) is 1 more than its enclosing type.

4.2 Scalar Inheritance Metrics

The first set of metrics are what we refer to as "scalar" metrics — they all produce a single scalar value for a user-defined type, such as the original DIT and NOC metrics do. All the metrics are defined in terms of paths (following edges) in the DAG and for the purpose of this paper we only present these metrics for classes or interfaces (that is, neither enums, annotations, nor exceptions). All paths consist of either all `extends` edges, in which case all the vertices are represent either classes or interfaces, or at most a single `implements` edge, which will have only vertices represent classes before it and vertices representing interfaces after it. In all cases, we do *not* count `java.lang.Object`.

There are roughly 4 categories of metrics — one category involves paths going from the type being measured to a root ("depth in tree" — DIT), one involves the number of other types reachable from the type either directly ("number of parents" — NOP) or transitively ("number of ancestors" – NOA), one is the number of other types from which the type being measured is reachable either directly ("number of children" — NOC) or transitively ("number of descendants" — NOD), and the last involves paths from a leaf to the type being measured ("height in tree" – HIT), however we do not consider this last category in this study.

Within each category, we can specify different metrics by specifying the allowable vertices and edges in the paths we consider. Some distinctions include: paths that only begin at classes and only follow `extends` edges ("CC"), paths that only begin at



DITCCUD(A)=0	DITCCUD(B)=1 NOAUD(B)=1	DITCCUD(C)=1 NOCCC(C)=2 NODCC(C)=3	DITCCUD(E)=3	DITIIUD(G)=1 NOCII(G)=2 NODII(G)=2 NOD(G)=5	DITIIUD(H)=2
NOCII(I)=1 NOCCI(I)=1 NOCI(I)=2 NOAUD(J)=4	DITIIUD(J)=2 NOAIIUD(J)=4 NOPI(J)=2	DITCCUD(K)=0	NOAUD(L)=7 NOPC(L)=2	NOPCI(N)=2 NOPC(N)=2	DITIIUD(P)=0 NOCCI(P)=2 NOAIIUD(P)=0

Fig. 1. Example of scalar metrics

interfaces and only follow `extends` edges ("II"), paths that only begin at a class and must begin with an `implements` edge ("CI"), paths that begin at classes and follow any edges ("C"), similarly for interfaces ("I"), or paths that begin at any type and follow any edges.

The name of a metric indicates its category and the kinds of edges allowed. Thus, for example, DITCC (DIT category, CC edges) is the length of the longest path starting at a class and following only `extends` edges to the root, NOCII is the number of interfaces incident (via `extends` edges) on an interface, and NOPCI is the number of interface vertices reachable from the type via an `implements` edge.

Following our discussion in section 3, we have two variants of DIT relating to whether or not the paths consist only of user-defined (UD) types or not. So, DITCCUD considers only paths that end at a non user-defined class. If the last class is `Object`, then DITCCUD is one less than the length of the path, otherwise it is the length of the path. DITCCUD is undefined for anything other than classes. DITIIUD is the equivalent for interfaces.

NOCI is a metric that applies only to interfaces and measures the indegree of the corresponding vertex. As we will see, this is useful as an interface with NOCI of zero is one that is neither implemented nor extended. NOPC only applies to classes and is the outdegree of the vertex. This tells us how many parents, following both `extends` and `implements` edges.

There are 2 NOA variants and 3 NOD variants. NOA has the same problem as DIT with respect to external libraries so we define metrics that refer only to user-defined types. NOAIIUD, NODCC, and NODII follow the conventions established

above (NOACCUD is equivalent to DITCCUD). NOAUD and NOD do not restrict the paths when determining what is an ancestor or descendant. NOAUD for a type X is then the number of variables with different user-defined types than X to which values of type X can be assigned. NOD for a type X is then the number of values with types different to X that can be assigned to a variable of type X.

Figure 1 gives examples of a number of the metrics. All metrics are also summarised in Appendix A.

4.3 Inheritance Summary Metrics

Inheritance Summary metrics apply to applications, that is, they produce values that are measurements of an application rather than an individual type as the scalar metrics do. These metrics report the proportion of *user-defined* types that fall into different categories. In the following, "DUI" (Defined Using Inheritance) denotes metrics that consider the types that occupy the child end of an edge in the inheritance DAG and "IF" (Inherited From) denotes metrics that consider types at the parent end. We can focus on what kinds of types participate in an inheritance relationship, with "CC" indicating class–class relationships (i.e., `extends`), "CI" indicating class–interface relationships (i.e., `implements`), "II" indicating interface–interface relationships (i.e., `extends`), and so on. As indicated above, we divide the user-defined types involved in an application into 3 subsets according to their origins, which we denote `UD` (user-defined), `TP` (third-party), and `SL` (standard library).

We begin with two metrics that give an overall idea of how much inheritance exists in an application.

DUI. The proportion of types that either implement an interface or extend another type other than `Object`, or, the proportion of types that occupy a child end of an edge in the inheritance DAG.

IF. The proportion of types that are either extended or implemented, or, the proportion of types that occupy a parent end of an edge in the inheritance DAG.

While these two metrics give us the proportion of user-defined types that participate in an inheritance relationship, we must keep in mind the interface/class distinction. For example, it seems reasonable to expect that all user-defined interfaces will be implemented, and thus boost the DUI measurement, so we have more refined metrics.

CCDUI. The proportion of user-defined *classes* that *extend* some other class.

CIDUI. The proportion of user-defined *classes* that *implement* some other interface.

IIDUI. The proportion of user-defined *interfaces* that *extend* some other interface.

CCIF. The proportion of user-defined *classes extended by* some other (user-defined) class.

CIIF. The proportion of user-defined *interfaces implemented by* some (user-defined) class.

IIIF. The proportion of user-defined *interfaces extended by* some other (user-defined) interface.

We can specify more refined metrics for the "DUI" category by classifying the types being extended. For each of the possible type relationships (table 1), we can consider the proportion of those relationships that have parents in `SL`, `TP`, or `UD`. We name these metrics by indicating which parent subset, which relationship, and the fact that we are measuring proportions at the child end of the relationship. So, for example, the proportion of classes that inherit from standard library classes is SLCCDUI, the proportion of classes that implement third-party interfaces is TPCIDUI, and the proportion of interfaces that extend user-defined annotations is UDIADUI.

Finally, we can specify metrics for types at a given nesting level, indicated by a subscript denoting the nesting level. So SLIIDUI_1 is the proportion of level-1 nested interfaces that extend standard library interfaces.

The metrics are also summarised in Appendix A.

5 Results

We have created a standard corpus of software to use for these kinds of studies [20]. For this study we analysed a total of 239 different codesets from 93 different open-source Java applications from the corpus. We list the latest version of each application in Appendix B. Considering only the latest version of each application, we measured 96,302 classes and 12,665 interfaces (108,967 types in total). The instrument we used for measuring looks at the bytecode version of the codeset.

In the previous section we described over 50 metrics (not counting nesting level distinctions). Due to space constraints, we present here just those measurements that seem most interesting. In particular, we provide only measurements relating to all classes and interfaces regardless of nesting level (leaving out those for enums, annotations, and exceptions). The full dataset is available on request.

5.1 Scalar Inheritance Metrics

We begin with the scalar metrics. Table 2 shows the maximum values we saw of each of these metrics, together with the applications that had types with those maximum measurements.

Some measurements are unsurprising, as are the applications that have the maximum measurements. For example, the maximum DITCCUD measurement is 10, which is consistent with other studies. Also, `eclipse` is one of the larger applications in our study (17622 classes, 1926 interfaces), and so it is unsurprising that it has many of the maximum values, although that one class has 795 children is noteworthy. Yet the much smaller `openoffice` (1320 classes, 1617 interfaces) has an interface with even more interface children. Trove's NOPCI (number of interfaces a class implements) value of 56 also seems rather extreme. The class with that value is `gnu.trove.SerializationProcedure`, which does not extend anything (other than `Object`) and so the NOPC maximum is the same.

Figure 2 shows frequency distributions for various tree depth metrics, summing all applications across the whole corpus, and reporting results in absolute values of metrics (x axis) for absolute number of classes with that metric value (y axis) on a log-log scale.

Table 2. Maximum values for scalar metrics

Metric	max	Applications
DITCCUD	10	netbeans-5.5-beta
DITIIUD	8	scala-1.4.0.3,netbeans-5.5-beta
NOCCC	795	eclipse_SDK-3.1.2-win32
NOCCI	279	eclipse_SDK-3.1.2-win32
NOCII	878	openoffice-2.0.0
NOCI	878	openoffice-2.0.0
NOPCI	56	trove-1.1b5
NOPC	56	trove-1.1b5
NOPI	13	luxor-1.0-b9
NODCC	983	eclipse_SDK-3.1.2-win32
NODII	1244	openoffice-2.0.0
NOD	1873	eclipse_SDK-3.1.2-win32
NOACCUD	11	netbeans-5.5-beta
NOAIIUD	18	glassfish-9.0-b15
NOAUD	57	trove-1.1b5

The first four graphs concern the "client" side of the inheritance relationship, that is, how a class relates to other classes that it inherits from. The first graph shows DITCCUD, that is the number of transitive superclasses (ancestors) of each class (not counting `Object`). The graph shows, for example, that over 10,000 classes have precisely 2 transitive superclasses not including `Object`, for example `Vector` extends `AbstractList` which extends `AbstractCollection` which extends `Object`, while only 100 classes across our corpus have 7 transitive superclasses. The second graph, DITIIUD, is similar to the first but for interfaces, and counts the length of the longest chain of transitive superinterfaces of each interface. The shape of the distributions are similar, except that there are far fewer interfaces than classes in the corpus — roughly one interface for every ten classes. The third graph, NOPC, shows the number of parents (classes and interfaces) that each class extends or implements; while the fourth graph, NOAUD, shows all ancestors, that is the transitive closure of all classes and interfaces contribution to a definition by any kind of inheritance.

These four client graphs have the same shape, which we have previously described in software as a "*truncated curve*" distribution [21]. Truncated curves are most likely log normal or stretched exponential distributions, and so quite different from normal distributions. Truncated curve distributions are highly skewed: almost every class will have a metric value of at least one, but this then decreases rapidly. Truncated curve distributions also have a maximum value (as their name suggests, they are truncated where they meet the x axis) that tends not to depend upon the size of the underlying data set — for depth (DITCCUD and DITIIUD) this is around 10; for parents (NOPC) 10 and for ancestors (NOAUD) around 12 for most applications. The maximum values for the parents and ancestors metrics are from `trove`, a library rather than an application, which is a clear outlier.

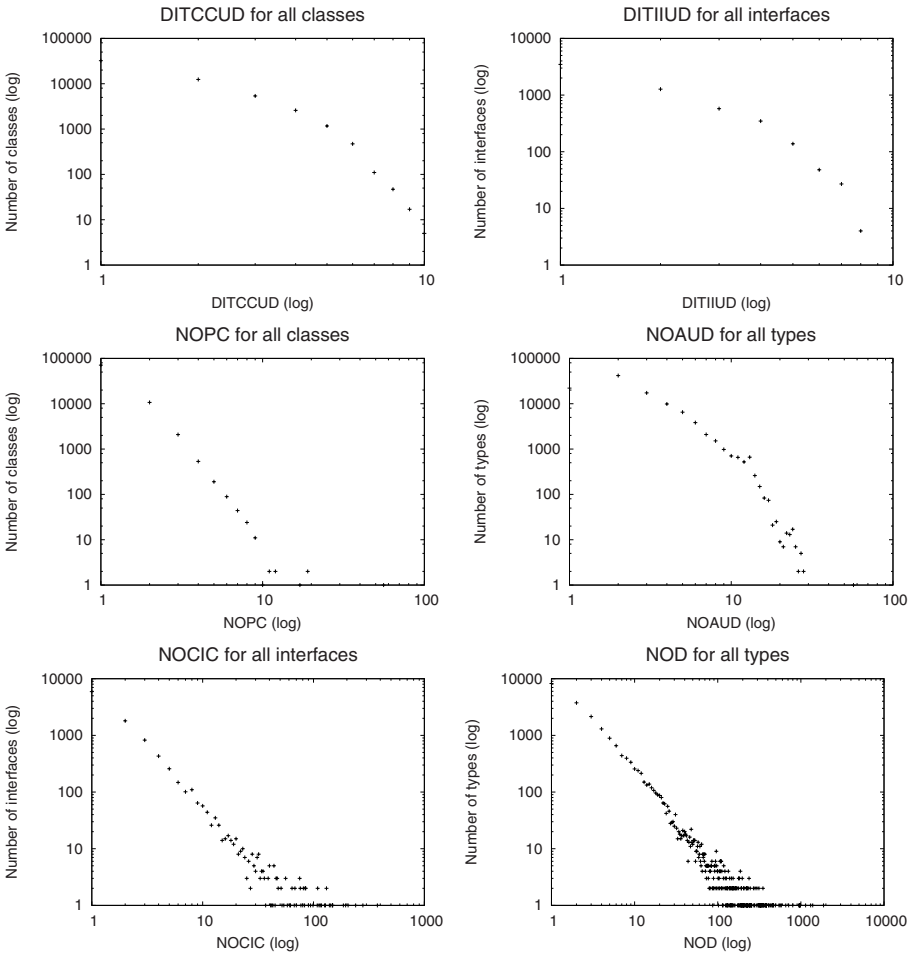


Fig. 2. Frequency distributions for scalar metrics over entire corpus (log-log)

These distributions mean that most inheritance is *shallow* — although most classes use inheritance, they are generally only a few levels down in the inheritance tree. Because of the skewed, truncated curve distributions, there will still be some classes that are quite deep in the inheritance hierarchy, but this is bounded — the number of ancestors contributing to each class' definition, and the depth of classes in the inheritance tree, does *not* increase with program size.

The other graphs in figure 2 concern the "supplier" side of the inheritance relationship, that is, how a class relates to other classes that inherit from it. The first supplier graph, NOCCI, shows for each interface, the number of classes that implement that interface (a partial complement to NOPC), while the second supplier graph, NOD, shows

the total number of types (classes and interfaces) that directly or transitively inherit from that class or implement that interface (a complement to NOA).

These two supplier graphs show the classic signature of a power-law distribution: a straight line on a log-log plot. In the centre of the data, a power law behaves quite similarly to a truncated curve, so that, for example, 1,000 classes implement 2 interfaces, but only 100 classes implement 10 interfaces (NOCCI); or 10,000 types have *no* descendants, 5,000 types have only one descendant, and around 100 types have 11 descendants (NOD). The key difference between a truncated curve and a power law occurs on the right hand-side — while the absolute metric values in a truncated curve distributions are, well, truncated, power laws are unbounded. This is visible towards the bottom right of the NOCCI and NOD graphs, which show that the corpus contains one or two classes with large values for these metrics, and in the maxima for the power law metrics (279 for NOCCI, 1895 for NOD, both from `eclipse`, one of the largest applications in our corpus).

Compared with normal distributions, power law distributions are counter-intuitive: most classes are not used as suppliers in inheritance relationships, while a few classes are used very commonly. Because power law distributions are unbounded, we can expect that as programs get larger, the numbers of implementations of popular interfaces and the number of descendants of popular classes will grow without limit.

In general, these graphs confirm the results of our smaller and much more coarse-grained study of general dependency topologies in software [21]: client relationships are truncated curves, while supplier relationships are power laws. Specifically with respect to inheritance, we see that most classes are defined using some form of inheritance — either extending another class or implementing at least one interface. Although inheritance is used pervasively to help define classes, it is also shallow: we found no class with more than ten parents, and very few with more twelve or thirteen ancestors (both classes inherited from, or interfaces implemented). For programmers reading or writing new class definitions, this means that they only need to consider a limited number of classes to understand their new classes.

On the other hand, relatively few classes and interfaces participate in the definitions of other types, but a few of those that do are used very widely indeed — this is the asymmetry inherent in power law and truncated curve networks. For programmers learning libraries or applications, this is good news: it means that there will be a few crucial types that they need to understand in order to implement or subclass to extend applications, or to use libraries. Furthermore, querying codesets to find the most frequently extended or implemented classes will likely be a good strategy to use when encountering an unfamiliar problem. On the other hand, for maintenance programmers, this is mixed blessing. Most types do not participate in much inheritance as suppliers, so they can be changed with little effect on the application. There will be some classes and interfaces, however, that are used very widely throughout a program, and as the program gets bigger (as more functionality, or more classes are added) these core classes and interfaces will be used more and more often. Maintaining or extending such classes or interfaces will be very difficult indeed, and only get harder as the size of programs increases.

5.2 Inheritance Summary Metrics

We now turn to the metrics that apply to whole applications. Figure 3 shows the DUI and IF results. It shows the number of applications having a given measurement, remembering that a measurement is a proportion of some kind. Looking at figure 3, the tallest bar (mode) for DUI is at 72% and has height 7, indicating that 7 applications had 72% of their types defined using inheritance in some way.

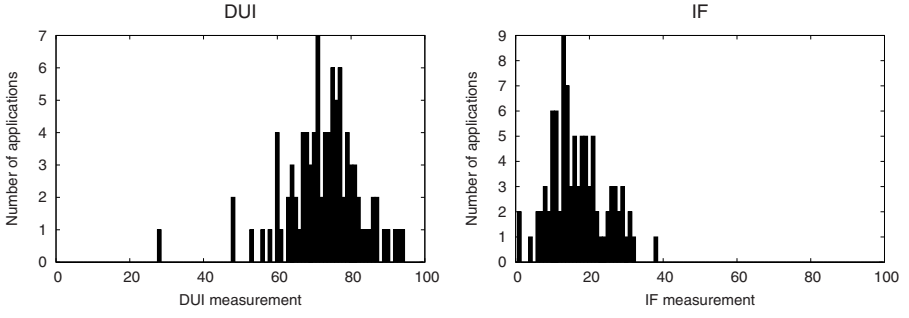


Fig. 3. Frequency distributions for DUI and IF

The striking feature of the DUI results is that the lowest is 29% (`mvnforum`), and that is very much an outlier. The next smallest value is 49% (`openxchange`, `quilt`). The median is 74%, that is, half the applications in our study have 74% or more of their user-defined types defined using some form of inheritance. For IF, the minimum was 2% (`ireport`,`rssowl`), the maximum was 39% (`scala`), the median was 17%, and mode was at 14% (9 applications).

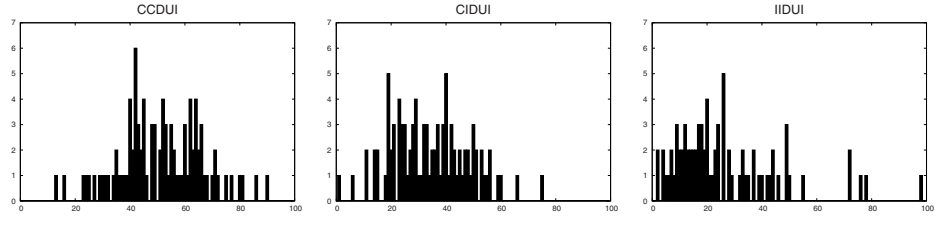


Fig. 4. Frequency distributions for CCDUI, CIDUI, and IIDUI

A class contributes to the DUI measurement by either extending another class or implementing an interface. The first is measured by CCDUI and the second by CIDUI. An interface contributes to DUI only by extending another interface (IIDUI). Their

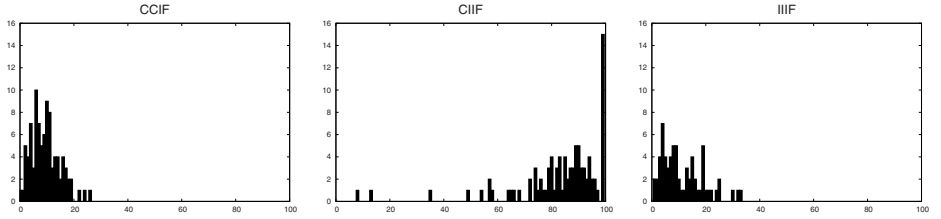


Fig. 5. Frequency distributions for CCIF, CIIF, and IIIF

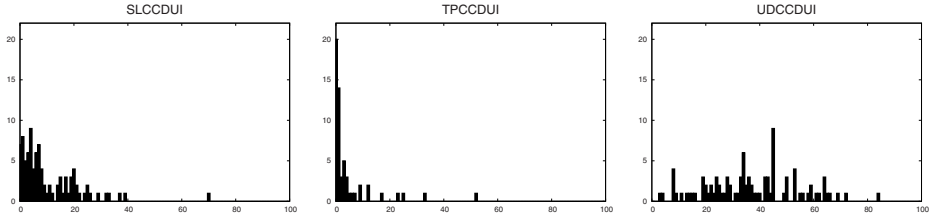


Fig. 6. Frequency distributions for SLCCDUI, TPCCDUI, and UDCCDUI

frequency distributions are shown in figure 4. For CCDUI, the minimum value is 14% (`mvnforum`), the median is 52%, the maximum is 91% (`jparse`), and the mode is 43% (6). For CIDUI, the minimum is 2% (`fitjava`), the median 34% , the maximum 76% (`scala`), and the mode 41% (5). Generally a lower proportion of classes inherit from an interface than extend another class. For IIDUI the minimum is 3% (roller), the median is 21% (`ganttproject`, `xalan`, `hibernate`, `lucene`), the maximum is 99% (`scala`), and the mode is at 27% (5).

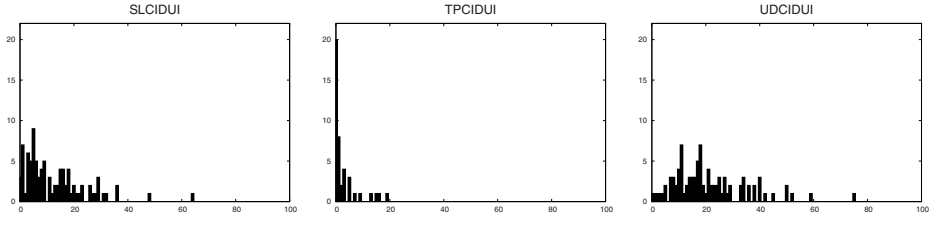


Fig. 7. Frequency distributions for SLCIDUI, TPCIDUI, and UDCIDUI

Considering figure 4, we can see that there was a wide distribution in the use of inheritance to define classes and interfaces across our corpus. So it is not the case that the distribution in figure 3 is due to (for example) just classes implementing interfaces, but each of CC, CI, and II relationships make significant contributions. In fact, applications

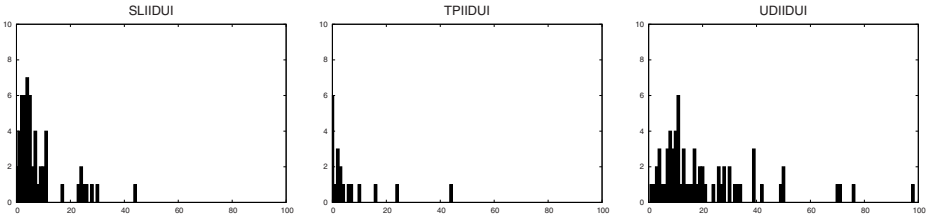


Fig. 8. Frequency distributions for SLIIDUI, TPIIDUI, and UDIIDUI

generally have a lower proportion of classes implementing interfaces than classes extending classes, and interfaces extending interfaces is lower still (note that a class both extending another class and implementing one or more interfaces will show in both the CCDUI and CIDUI results).

A type contributes to the IF measurement by either being an interface that is implemented (CIIF) or extended (IIIF) or a class that is extended (CCIF). Their frequency distributions are shown in figure 5. We see that rarely are more than 20% of either classes or interfaces extended, and most of the time more than 80% of interfaces are implemented. The latter is somewhat surprising — we would expect that if an interface is created then it would be implemented, but this is true for only 15 of the applications. One possible explanation is that some interfaces are only extended, hence it is worth looking at how many children an interface has (NOCI). The lowest CIIF value is 9% for `openoffice`, but recall that `openoffice` had the largest NOCI value.

A class can extend either a standard library class (SLCCDUI), a third-party class (TPCCDUI), or a user-defined class (UDCCDUI). Figure 6 shows their distributions. It would appear that most of the CCDUI distribution can be explained by the UDCCDUI distribution, that is, by and large, classes that extend another class tend to extend user-defined classes. Figures 7 and 8 shows the distributions for the CI and II relationships.

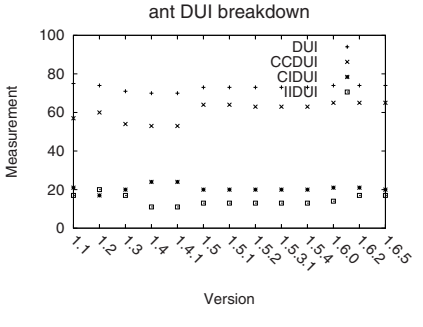


Fig. 9. DUI and its breakdown for 13 versions of ant

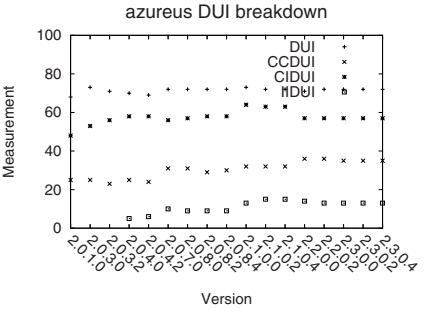


Fig. 10. DUI and its breakdown for 17 versions of azureus

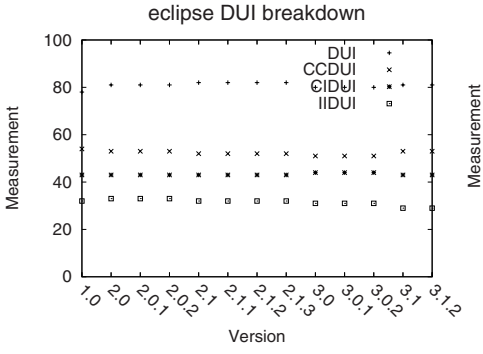


Fig. 11. DUI and its breakdown for 13 versions of eclipse

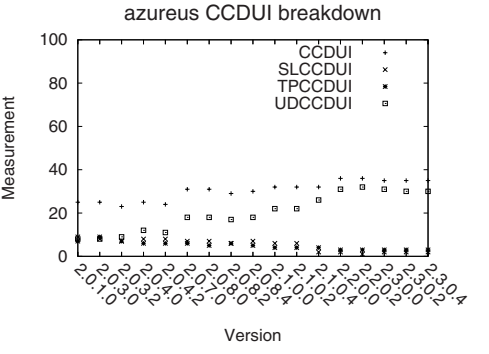


Fig. 12. CCDUI and its breakdown for 17 versions of azureus

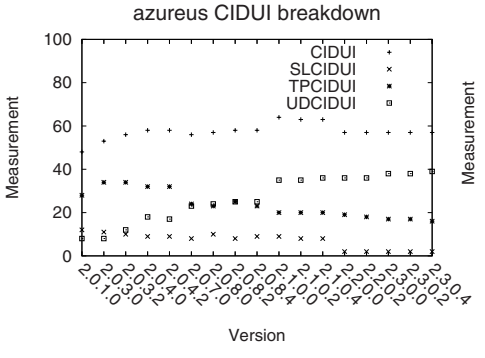


Fig. 13. CIDUI and its breakdown for 17 versions of azureus

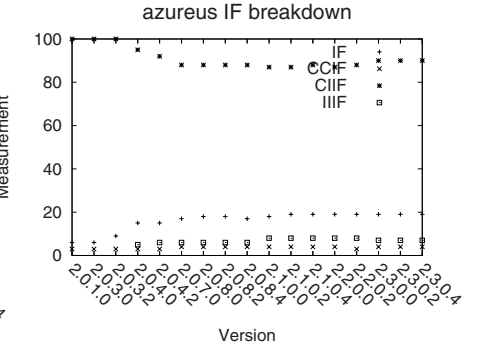


Fig. 14. IF and its breakdown for 17 versions of azureus

5.3 Longitudinal

Another view of our data is to consider how the various metrics change over time for an application. Figures 9, 10, and 11 show DUI and its breakdown (CCDUI, CIDUI, IIDUI) on one chart each for `ant`, `azureus`, and `eclipse` respectively. What is particularly interesting about these figures is how consistent the values are, especially given the changes in size of the three applications: the number of user-defined types for `ant` goes from 102 for `ant-1.1` to 1014 for `ant-1.6.5`, for `azureus` it is from 163 for `azureus-2.0.1.0` to 2130 for `azureus-2.3.0.4`, and for `eclipse` it is from 6522 for `eclipse_SDK-1.0-win32` to 19674 for `eclipse_SDK-3.1.2-win32`.

Figures 12 and 13 show the longitudinal views of CCDUI and CIDUI for `azureus`. We note that the x-axis is not a linear scale, and that generally the size (in number of types) of `azureus` increases over the period shown. For example, for SLCCDUI, the earliest version has 15 classes whereas the latest has 42, so the absolute number

has increased but the proportion is declining. On the other hand, UDCCDUI shows a marked increase, from 8%, 13 out 158 (version 2.0.1.0) to 27%, 647 out of 2410 (version 2.3.0.4).

Figure 14 shows the IF results and the breakdown for `azureus`. IF shows an initial increase and then an apparent convergence to 19%. CCIF ends up holding steady at 4% from about version 2.0.7.0, but that version has 860 classes and the final version has 2410. As a general rule we would expect all interfaces to be implemented (CIIF). As we noted earlier, reasons for this not to be the case include interfaces that exist only to be extended by others, or dead code. In this case only around 90% of interfaces are directly implemented, but there were only 5 in the earliest version and 492 (444 implemented) in the latest. As there are 492 interfaces and 2410 classes in the latest version, giving 2902 types overall, the fact that 444 are implemented accounts for 15% overall, or most of the IF measurement (19%). However, from the IIIF data we find that 35 interfaces appear not be used at all.

6 Discussion

Our study has revealed several interesting features about the way inheritance is actually used in practice in Java programs. We find the DUI results particularly interesting: as presented in section 5.2, ***around three-quarters of user-defined classes use some form of inheritance*** in at least half the applications in our corpus. We expected to see much lower proportions of types using some form of inheritance. Our first thought on seeing these results was that the high values could be due to heavy use of interfaces, or perhaps significant use of frameworks from the standard API or third party libraries. Refinement of our measurements (figures 4 and 6) show that neither are the case, instead the most common (but by no means the predominant) form of inheritance is classes extending other user-defined classes.

One possible explanation of our observations is that some amount of the inheritance we see is "bad inheritance" in some sense. It could be that there is so much advice around advocating avoiding inheritance where possible because it is mainly being used inappropriately. We cannot rule this out, although various checks we performed on the accuracy of our tool involved looking at the source code, and these casual observations did not reveal anything obviously wrong with the use of inheritance we saw, and our corpus is made up of well-known applications mostly written by professional Java programmers. So we interpret our data to mean that defining most classes by inheritance is accepted practice in Java programming.

Our observations thus provide a useful benchmark for managers of Java programmers — applications with significantly less than 75% of types defined using inheritance, or more than 17% are inherited from, or in other ways significantly differ from what we have observed, are applications that probably need investigation.

While a large proportion of types are defined using inheritance, rarely are more than 20% of classes or interfaces extended. While we have not shown the figures here, it also turns out that the proportion of interfaces is rarely more than 20% of the total types, and usually around 10%. This points towards a significant fan-out in the inheritance relationships, and the NOC category of scalar metrics supports this view.

We have noted the NOC frequency distributions appear to have a classic *power-law* shape we have seen in other software relationships [22,21]. If the NOC metrics do have a power-law distribution, then it follows that the larger the application, the larger the NOC values we will see, and statistical measures such as mean and standard deviations will have no useful purpose. This may explain why our NOC results are so different from the previously published data — our study is so much larger.

Other distributions such as the NOD and NOP categories also appear to be power-laws (although the NOP could be an artifact due to the small number of data points). We have also noted the appearance of the "truncated curve" distribution (figure 2) we have observed in dependency metrics in a previous study [21].

Examination of the longitudinal results also reveals a surprising feature, namely how constant the use of inheritance is across application evolution and application size. While 3 data points is hardly a strong trend, the fact that any exist is noteworthy. There is no obvious reason for why such consistency should occur. Given the significant changes in size (for example `ant` grows an order of magnitude across the versions we studied), it seems unlikely that it is due to something about the delivered functionality, as the delivered functionality will have changed significantly. It is possible that this application is biased by some feature of its problem domain, or the programming style used by the development team, however it is equally likely that this level of use of inheritance is simply a standard feature of accepted Java programming practice.

Another interesting feature is how much the component parts of DUI vary despite DUI itself being so constant. For example in `azureus` (figure 10) the proportion of classes implementing interfaces noticeably increases and then decreases without a change in DUI and without similar differences in the proportion of classes or interfaces using inheritance.

Looking at the breakdown of CCDUI for `azureus` (figure 12) we see that the proportion of classes extending other user-defined classes steadily increases while the proportion extending standard library or third-party classes steadily decreases, while (not shown in the figure) the overall size of `azureus` grows (although we again note that the x-axis is not a linear scale). This suggests that applications become more inwards looking as they age, relying on their own definitions. If externally-provided functionality is required (by inheritance), it may be more likely to be accessed via user defined classes that either themselves inherit or delegate to external code.

We note that the largest application in our study (`netbeans`, 19666 classes and 1830 interfaces) only has one maximum scalar measurement, and several different applications of quite different sizes are represented in the maximum scalar metric measurements. This suggests that our metrics are not simply measuring application size (at least as measured by number of types), but are actually capturing other features of programming style or practice.

There are many other points of interest in the data we have collected that we do not have space to discuss. But for example, we mentioned earlier interfaces that are neither implemented nor extended (NOCI measurement of 0). There are in fact over 2000 such interfaces. Some provide only constants, some indicate variation points of frameworks, and some seem to be just dead code. This raises the question as to how much dead code

is being distributed. We also wonder what other peculiarities we might find in our data, and in other kinds of similar measurements that could be made of code.

The most likely 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 (`openoffice`, `eclipse`, `ant`) are some of the most used Java programs worldwide. It does however raise the question as to whether our results indicate something specific to the open-source development model. We note that the few other similar studies that have been published [11,7,8,9,12,13,16] generally indicate different results, although their small size, the lack of data they present, the lack of clarity about what they are measuring, and the coarse granularity of their metrics makes it difficult to tell. In the scientific tradition, we hope we have provided sufficient details about our corpus and metrics to allow other researchers to replicate our study: independent replication will give the best grounds to claim generalisability.

7 Conclusions

Like all programming language designs, Java is an experiment. Unlike most language designs, the general adoption of Java, and the resulting widespread availability of substantial "real-world" Java programs means that we are finally able to evaluate that experiment, in ways that are simply not possible for most other languages.

In this paper, we have introduced a new structured suite of metrics to evaluate, quantitatively, how Java programs use inheritance. More importantly, we have applied these and some more traditional "scalar" metrics in a large-scale empirical study. We believe such studies are important to establishing and understanding trends in software development.

Our results show surprisingly high levels of use of inheritance in defining types, with about 3 out of 4 types in our study being defined using inheritance in one form or other. In contrast, most types make only a small contribution to other definitions via inheritance; however a few types will be very well used, being inherited or implemented by many other classes or interfaces. We have also seen evidence that levels of inheritance are somewhat constant over the lifetime of an application. Our corpus study indicates that an apparently high use of inheritance is a characteristic of accepted Java programming practice.

The overarching methodological contribution implied by our results is that metrics for inheritance must distinguish between classes and interfaces, and between extends and implements relationships. To do otherwise obscures important data about program structure, because our results show that different kinds of inheritance are used in different ways. Furthermore, distinguishing between user code and "other" code (both standard libraries or third party components) is also important to give a true picture of the use of inheritance. So far, our results show that programmers treat standard libraries and third-party code in the same way — at least as far as inheritance is concerned — while user-defined classes are treated differently, primarily by being used more often to define other user-defined classes.

We emphasise that we make no claims as to whether our results are indicative of "good design". Without reliable data about such things as development effort, presence

of faults, and other quality attributes, we cannot make such an assessment. As others before us have observed, gathering such data is crucial to understanding the impact of such things as inheritance structure on software quality. The contribution of this research is a crucial prerequisite to doing such studies: first, being able to understand and measure the various uses of inheritance in Java programs in a well-founded manner; and second, being able to use those measures to quantify the accepted Java programming practice.

There are many directions this work can take. We have collected, but have not yet analysed, data on the use of nested classes, including static nested classes, in inheritance relationships. Java also distinguishes abstract from concrete classes, and it would be interesting to determine how they are used. Others have discussed examining the number of methods inherited, and other such "internal" inheritance relationships [23,12]. Steimann has identified various roles that interfaces can play, and extending his study to our corpus may prove interesting [24]. As we noted at the end of the previous section, independent replication of our results would give more support for generalisation across other Java programs. Replication of our studies in other OO languages would help determine how much our results depend on features Java, and how much they are in some sense intrinsic to object-orientation.

Given enough data, it is often possible to find some kind of pattern, and we certainly have plenty of data. Nevertheless we suggest that the patterns we have observed are indeed an indication of significant structures in software design, and faithfully capture large-scale aspects of the use of inheritance in accepted Java programming practice. In other words, we have shown how Java programs use inheritance.

Acknowledgements

We would like to thank the anonymous referees for their comments and suggestions for improving this paper and the suggestions for new studies.

References

1. Taivalsaari, A.: On the notion of inheritance. Comp. Surv. 28(3), 438–479 (1996)
2. Meyer, B.: Reusability: the case for object-oriented design. IEEE Software, 50–64 (March 1987)
3. Snyder, A.: Inheritance and the development of encapsulated software components. In: Research Directions in Object Oriented Programming, pp. 165–188. MIT Press, Cambridge (1987)
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison Wesley Publishing Company, One Jacob Way, Reading, Massachusetts 01867 (1994)
5. Johnson, R.E., Foote, B.: Designing reusable classes. Journal of Object-Oriented Programming (June/July 1988)
6. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley, Reading (2004)
7. Daly, J., Brooks, A., Miller, J., Roper, M., Wood, M.: Evaluating inheritance depth on the maintainability of object-oriented software. Empirical Software Engineering 1(2), 109–132 (1996)

8. Cartwright, M.: An empirical view of inheritance. Information and Software Technology 40, 795–799 (1998)
9. Harrison, R., Counsell, S., Nithi, R.: Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. Journal of Systems and Software 52, 173–179 (2000)
10. Chidamber, S.R., Kemerer, C.F.: Towards a metrics suite for object oriented design. In: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 197–211 (1991)
11. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20(6), 476–493 (1994)
12. Manel, D., Havanas, W.: A study of the impact of C++ on software maintenance. In: International Conference on Software Maintenance, pp. 63–69 (1990)
13. Succi, G., Pedrycz, W., Djokic, S., Zuliani, P., Russo, B.: An empirical exploration of the distributions of the Chidamber and Kemerer object-oriented metrics suite. Empirical Softw. Engg. 10(1), 81–104 (2005)
14. Collberg, C., Myles, G., Stepp, M.: An empirical study of Java bytecode programs. Softw. Pract. Exper. 37(6), 581–641 (2007)
15. Chidamber, S., Darcy, D., Kemerer, C.: Managerial use of metrics for object-oriented software: an exploratory analysis. IEEE Trans. Software Engineering 24(8), 629–639 (1998)
16. Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. 22(10), 751–761 (1996)
17. Briand, L.C., Daly, J., Porter, V., Wüst, J.K.: A comprehensive empirical validation of design measures for object-oriented systems. In: METRICS 1998: Proceedings of the 5th International Symposium on Software Metrics, pp. 246–257. IEEE Computer Society Press, Los Alamitos (1998)
18. Kitchenham, B., Pfleeger, S.L., Fenton, N.: Towards a framework for software measurement validation. IEEE Trans. Softw. Eng. 21(12), 929–944 (1995)
19. Melton, H., Tempero, E.: An empirical study of cycles among classes in Java. Empirical Software Engineering 12(4), 389–415 (2007)
20. Qualitas Research Group: Qualitas corpus (June 2007), `http://www.cs.auckland.ac.nz/~ewan/corpus/`
21. Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., Tempero, E.: Understanding the shape of Java software. In: Cook, W. (ed.) ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, Portland, OR, U.S.A, October 2006, pp. 397–412 (2006)
22. Potanin, A., Noble, J., Frean, M., Biddle, R.: Scale-free geometry in OO programs. Commun. ACM 48(5), 99–103 (2005)
23. Benlarbi, S., Melo, W.L.: Polymorphism measures for early risk prediction. In: ICSE 1999: Proceedings of the 21st international conference on Software engineering, pp. 334–344. IEEE Computer Society Press, Los Alamitos (1999)
24. Steimann, F., Mayer, P.: Patterns of interface-based programming. Journal of Object Technology 4(5), 75–94 (2005), `http://www.jot.fm/issues/issue_2005_07/article1`

Appendix A: Metric Summaries

Scalar Metrics

These metrics provide measurements for individual types in alphabetical order. In these definitions, A is an *ancestor* of X if, A is not `Object` and there is a path from X to A

where all the vertices, with the possible exception of A, are user-defined types, and D is a *descendent* of Y if there is a path from D to Y.

DITCCUD. Length of path from a class and consisting only of `extends` edges to the first non user-defined class other than `Object`, or one less than the length of the path that ends with `Object`.
DITIIUD. Length of path from an interface and consisting only of `extends` edges to the first non user-defined class.
NOACCUD. Number of ancestors a class has (`extends` edges only).
NOAIIUD. Number of ancestors an interface has (`extends` edges only).
NOAUD. Number of all ancestors a type has (both `implements` and `extends` edges).
NOCCC. Number of classes inheriting from a given class (via `extends` edges).
NOCCI. Number of classes implementing a given interface (via `implements` edges).
NOCII. Number of interfaces inheriting from a given interface (via `extends` edges).
NOCI. Total number of classes that implement a given interface and interfaces that extend that interface (both `implements` and `extends` edges).
NODCC. Number of descendants a class has (`extends` edges only).
NODII. Number of descendants an interface has (`implements` edges only).
NOD. Number of all descendants a type has (both `implements` and `extends` edges).
NOPCI. Number of interface parents a class has (via `implements` edges).
NOPC. Number of parents a class has (both classes and interfaces).
NOPI. Number of parents an interface has (via `extends` edges).

Summary Metrics

These metrics provide measurements over an application. The first set (given in alphabetical order) do not consider the source of the types participating in a given relation (see the table below for that).

CCDUI. The proportion of user-defined *classes* that *extend* some other class.
CCIF. The proportion of user-defined *classes extended by* some other (user-defined) class.
CIDUI. The proportion of user-defined *classes* that *implement* some other interface.
CIIF. The proportion of user-defined *interfaces implemented by* some (user-defined) class.
DUI. The proportion of types Defined Using Inheritance, that is, those types that either implement an interface or extend another type other than `Object`, or, the proportion of types that occupy a child end of an edge in the inheritance DAG.
IF. The proportion of types Inherited From, that is, those types that are either extended or implemented, or, the proportion of types that occupy a parent end of an edge in the inheritance DAG.
IIDUI. The proportion of user-defined *interfaces* that *extend* some other interface.
IIIF. The proportion of user-defined *interfaces extended by* some other (user-defined) interface.

The table below lists the most refined summary metrics. Each row is one of the 7 relationships identified (whether it is `extends` or `implements` is implied by the combination of kinds of type). The columns show the two directions of the relationship. The cells of the "using" relation have the metrics in the order of: using **S**tandard **L**ibrary, using **T**hird **P**arty, or using **U**ser **D**efined.

	Defined Using Inheritance (Using)	Inherited From (Used)
Class-Class	SLCCDUI, TPCCDUI, UDCCDUI	CCIF
Class-Interface	SLCIDUI, TPCIDUI, UDCIDUI	CIIF
Interface-Interface	SLIIDUI, TPIIDUI, UDIIDUI	IIIF
Interface-Annotation	SLIADUI, TPIADUI, UDIADUI	IAIF
Enum-Interface	SLEIDUI, TPEIDUI, UDEIDUI	EIIF
Exception-Interface	SLExIDUI, TPExIDUI, UDExIDUI	ExIIF
Exception-Exception	SLExExDUI, TPExExDUI, UDExExDUI	ExExIF

Appendix B: Applications from Qualitas Corpus

We believe it is important to provide as complete information as possible regarding the applications used in our study, although space constraints cramp our presentation somewhat. The format is *application name-version id*.
aglets-2.0.2, ant-1.6.5, antlr-2.7.6, aoi-2.2, argouml-0.20, axion-1.0-M2, azureus-2.3.0.4, c_jdbc-2.0.2, colt-1.2.0, columba-1.0, compiere-251e, derby-10.1.1.0, displaytag-1.1, drawswf-1.2.9, drjava-20050814, eclipse_SDK-3.1.2-win32, exoportal-v1.0.2, findbugs-1.0.0, fitjava-1.1, fitlibraryforfitnesse-20050923, freecol-0.6.0, freecs-1.2.20060130, galleon-1.8.0, ganttproject-1.11.1, geronimo-1.0-M5, glassfish-9.0-b15, gt2-2.2-rc3, heritrix-1.8.0, hibernate-3.1-rc2, hsqldb-1.8.0.4, htmlunit-1.8, infoglue-2.3Final, informa-0.6.5, ireport-0.5.2, itext-1.4, ivatagroupware-0.11.3, j_ftp-1.48, jag-5.0.1, jaga-1.0.b, james-2.2.0, jasperreports-1.1.0, javacc-3.2, jboss-4.0.3-SP1, jchempaint-2.0.12, jedit-4.2, jeppers-20050607, jetty-5.1.8, jfreechart-1.0.1, jgraph-5.9.2.1, jhotdraw-6.0.1, jmeter-2.1.1, joggplayer-1.1.4s, jparse-0.96, jrat-0.6, jrefactory-2.9.19, jspwiki-2.2.33, jtopen-4.9, jung-1.7.1, junit-4.1, log4j-1.2.13, lucene-1.4.3, luxor-1.0-b9, megamek-2005.10.11, mvnforum-1.0-ga, nekohtml-0.9.5, netbeans-5.5-beta, openjms-0.7.7-alpha-3, openoffice-2.0.0, openxchange-0.8.0.6, oscache-2.3-full, pmd-3.3, poi-2.5.1, proguard-3.6, quartz-1.5.2, quickserver-1.4.7, quilt-0.6-a-5, roller-2.1.1-incubating, rssowl-1.2, sablecc-3.1, sandmark-3.4, scala-1.4.0.3, sequoiaerp-0.8.2-RC1-all-platforms, servicemix-3.0-SNAPSHOT, soot-2.2.3, springframework-1.2.7, squirrel_sql-2.4, struts-1.2.9, tomcat-5.5.17, trove-1.1b5, webmail-0.7.10, xalan-j_2_7_0, xerces-2.8.0, xmojo-5.0.0.