An Empirical Study of Evolution of Inheritance in Java OSS

E. Nasseri, S. Counsell and M. Shepperd
School of Information Systems, Computing and Mathematics,
Brunel University, Uxbridge, Middlesex, UB8 3PH.
+44 (0)1895 266740
{emal.nasseri, steve.counsell, martin.shepperd}@brunel.ac.uk

ABSTRACT

Previous studies of Object-Oriented (OO) software have reported avoidance of the inheritance mechanism and cast doubt on the wisdom of 'deep' inheritance levels. From an evolutionary perspective, the picture is unclear - we still know relatively little about how, over time, changes tend to be applied by developers. Our conjecture is that an inheritance hierarchy will tend to grow 'breadth-wise' rather than 'depth-wise'. This claim is made on the basis that developers will avoid extending depth in favour of breadth because of the inherent complexity of having to understand the functionality of superclasses. Thus the goal of our study is to investigate this empirically. We conduct an empirical study of seven Java Open-Source Systems (OSSs) over a series of releases to observe the nature and location of changes within the inheritance hierarchies**.** Results show a strong tendency for classes to be added at levels one and two of the hierarchy (rather than anywhere else). Over 96% of classes added over the course of the versions of all systems were at level 1 or level 2. The results suggest that changes cluster in the shallow levels of a hierarchy; this is relevant for developers since it indicates where remedial activities such as refactoring should be focused.

Keywords: Object-oriented, Inheritance, Java, OSS.

1. INTRODUCTION

In this paper, we investigate the evolution of seven Open-Source Systems (OSSs) and the trends in inheritance hierarchies therein. This is of significance for two reasons. Firstly, if we can predict the most change prone parts of a system then we can pre-emptively target refactoring activity to such parts of a system. Secondly, it may yield information as to how software engineers view and understand complex legacy systems. The research problem is: how do inheritance hierarchies in object-oriented (OO) software systems evolve over time? More specifically, we conjecture that change will not be evenly distributed but will tend to cluster around the top levels (closer to the root) of such structures. We therefore conduct an empirical investigation to assess whether this is indeed the case. We focus on OSSs because OSS is becoming increasingly prevalent in commercial organisations and is the subject of continued research interest [1, 7, 9]. Moreover, we know of no study that has yet investigated the evolution of inheritance structures from an OSS perspective.

The original claim for using inheritance was that it modelled data in a structured and logical fashion, thus aiding the maintenance process [6]. Use of inheritance is claimed to reduce the amount of software maintenance necessary, ease the burden of testing [4], and produce more reliable, high quality software [2, 3]. While in theory this may make sense, from a practical perspective there is empirical research to suggest that deep levels of inheritance impede the maintenance process because of the comprehension overhead of needing to understand relevant super-classes [14, 18]. Given the dominance of OO technology over the past decade or longer it is not unsurprising that it has been the target of a good deal of empirical research, much of which has endeavoured to explore the extent to which the claims of OO proponents are well founded. A surprising finding is that the inheritance that was at least initially seen as a central aspect of the paradigm seems to be used little in practice. (We consider the evidence in more detail in the next section.) Another aspect that has not, to the best of our knowledge, been studied is the relationship between the inheritance structure and where maintenance activities occur.

The motivation for the research in this paper stems from a number of sources. Firstly, we know very little about how inheritance structures evolve over time [16, 17, 23]; the research in this paper seeks to shed light upon this issue. There is evidence to suggest that developers may find inheritance difficult to comprehend beyond a specific level. If that is true, then we would expect developers to add classes at shallow levels of the inheritance hierarchy rather than at deep levels. We posit that growth will be breadth-

wise not depth-wise, thus supporting a growing belief about the use of inheritance. We believe that a better understanding of the change behaviour, and in particular the locality, would enable refactoring resources to be targeted more efficiently. Secondly, we believe that a first-step towards a change prediction model is an appreciation of current trends in changes made to an inheritance hierarchy. Given that this is a resource intensive activity this would clearly be of benefit to software engineers (and potentially users) since the outcome could be more flexible and responsive software systems.

The remainder of this paper is organised as follows. In the next section we describe related work. In Section 3 we describe the study details including a description of the systems under investigation and the metrics collected. In Section 4, we analyse the data extracted. We then present a discussion of the results (Section 5) before drawing conclusions and pointing to future work (Section 6).

2. RELATED WORK

As indicated in the introduction, OO systems have been extensively researched from an empirical perspective. One area has been the drive to try to quantify different properties of such systems. For example, the Depth of Inheritance Tree (DIT) metric of Chidamber and Kemerer (C&K) [11] has been used extensively in empirical studies; the Specialization and Reuse Ratios proposed by Henderson-Sellers [19] have also featured in empirical studies.

There is some evidence to suggest that systems without inheritance (i.e. flat systems) are easier to modify and maintain than systems containing inheritance. Daly et al. [14] describe an experiment in which subjects were timed performing maintenance tasks on OO systems of varying levels of inheritance. Systems with 3 levels of inheritance were shown to be easier to modify than systems with no inheritance. Systems with 5 levels of inheritance were, however, shown to take longer to modify than the systems without inheritance. Harrison et al. [18] replicated the experiment and found that flat systems (containing no inheritance) were easier to modify than systems containing three or five levels of inheritance, although results indicated that larger systems were equally difficult to understand whether or not they contained inheritance. The multi-method study of Wood et al., [25] suggests that inheritance should be used with care and only when needed. Finally, two controlled experiments by Prechelt et al. [24] found that it took longer to maintain a program with higher levels of inheritance than a program containing fewer inheritance features.

In terms of other directly related work, various empirical investigations have been made into the use of inheritance. For example, the seminal paper by C&K describes their metrics [11] and detail empirical analyses of systems at two sites, one of which used C++ and the other Smalltalk. The extent of inheritance at both sites was small (with median Depth of Inheritance Tree (DIT) values of 1 and 3 for the C++ and Smalltalk sites, respectively). The explanation given is that designers wanted to retain comprehensibility and simplicity in favor of reuse. In Chidamber et al., [13], three commercial OO systems were empirically investigated, and, again, none showed significant use of inheritance. Bieman and Zhao [5] describe a study of 19 C++ systems, containing 2,744 classes in total. They found that only 37% of these systems had a median class inheritance depth greater than 1. Cartwright and Shepperd [10] describe the collection of a subset of metrics from a large telecommunications system (133,000 lines of C++). Their main finding was a positive correlation between the DIT metric of C&K [11] and number of user-reported problems, casting doubt on the effective use of inheritance. They also report relatively little use of inheritance in the system they analyzed. In Basili et al., [3] the results of an empirical study of the C&K metrics are presented. The metrics are used as predictors of fault-prone classes. Data from eight medium-sized management systems, developed in C++ was collected. An experimental hypothesis suggested that a class located deep in the inheritance hierarchy was more fault-prone than a class higher up in the hierarchy; this hypothesis was found to be supported with statistical significance. This clearly implies that, far from aiding maintenance, use of inheritance had the opposite effect.

Our claim, based on results from previous studies about the use of inheritance, is thus that inheritance hierarchies will tend to evolve on a 'breadth-wise' rather than 'depth-wise' basis thus giving the hierarchy a 'flattened' shape; the claim is based on the belief that rather than try to understand existing functionality of a hierarchy, developers will add classes at shallow levels instead. In other words, we believe that the original claim and purpose of inheritance is an impediment to developers when maintaining Java software and they will act accordingly when maintaining code. Reported results support our claim; we found the vast majority of added classes to be those as shallow levels of the hierarchy and relatively small activity at lower levels of the hierarchy. Such a trend may have significant implications for the location of faults in the short and long-term.

3. STUDY DETAILS

3.1 The Seven Open-Source Systems

The seven OSSs on which our study is based included a computer game and game engine, a template engine, a compiler construction tool, an SQL database, a documentation support and PDF file manipulation system. The systems were chosen sequentially from the range of systems available at sourceforge.net ensuring that 1) as wide a range of applications was chosen for external validity of the study *and* 2) a sufficient number of versions

were available of each system. Moreover, five of the seven systems were also used in a previous empirical study [1] and to allow comparison of results and further possible replication, we retained these same five systems (only JBoss and JAG were added in the study described in this paper). Each system thus comprised multiple versions and inheritance metrics were collected from each version. (We note that the 'final' version represents the latest version available to download and not the end version of the system). The systems studied (in ascending order of number of versions) were as follows.

1) HSQLDB: a relational database engine implemented in Java. This system comprised 6 versions. HSQLDB started with 65 classes in first version and comprised 358 classes by the final version.
2) JasperReports: a business intelligence and reporting engine. This system comprised 12 versions. JasperReports started with 818 classes and comprised 1098 classes by the final version.
3) EasyWay: a 2D Java game engine. This system comprised 21 versions. EasyWay started with 183 classes and comprised 197 classes by final version.
4) SwingWT: an implementation of the Java Swing and AWT APIs. This system comprised 22 versions. SwingWT started with 50 classes in its 1st version and increased in size to 620 by the final version.
5) JAG: Java Application Generator. Generates working projects containing complete J2EE applications. This system comprised 23 versions. JAG started with 137 classes and contained 136 classes by the final version.
6) JBoss: a standards-compliant, J2EE based application server implemented in Java. 27 versions of this system were available starting from version 8. JBoss was the largest system in size. It contained 3934 in version 8 and variably evolved. JBoss ended with 9082 classes by the final version.
7) Tyrant: a graphical fantasy adventure game. 45 versions of this system were studied. Tyrant started with 122 classes and finally ended with 273 classes by the final version.

3.2 Data collected

For this study we used an automated tool to collect four inheritance-based measures from each version of the seven systems. The JHawk tool is an OO metrics extractor, information about which can be accessed from:

(http://www.virtualmachinery.com/jhawkprod.html).

The four inheritance metrics collected were as follows.

1) Depth of Inheritance Tree (DIT): this metric measures the number of ancestors of a class including 'Object' from which all classes inherit. The DIT metric was proposed by C&K [11]. We assume the value of DIT for class 'Object' at the root of the entire hierarchy is zero; hence, all classes declared at level 1implicitly *extend* only class 'Object'.
2) Specialization Ratio (SR): this metric is calculated as: number of subclasses/number of superclasses. High values of the SR metric imply high level of reuse through subclassing [19].
3) Reuse Ratio (RR): this metric measures inheritance using the formula: number of superclasses/total number of classes. The total number of classes refers to total number of classes residing in inheritance hierarchy excluding class 'Object' [19]. An RR Value close to 1 implies that the inheritance hierarchy is *narrow*. An RR value close to 0 implies that the inheritance hierarchy is *shallow*.
4) Number of Children (NOC) metric: this measures the number of immediate subclasses of a class and was first proposed by C&K [11].

The four inheritance measures were collected from classes of each version of the seven systems. Note that we refer to a single 'inheritance hierarchy' of Java throughout the paper, since in Java every class inherits from Object. This is distinct from C++ where a class need not necessarily inherit from any other class or be inherited from. We also make no distinction between concrete and abstract classes for the purposes of our analysis.

3.3 Summary data

Table 1 shows, for each of the seven systems, in order of versions studied, the maximum (Max), minimum (Min), median (Med) and Mean change values in the number of classes across the versions studied. By 'change' we mean positive or negative 'growth' by either adding or deleting classes.

For maximum changes, Table 1 also indicates the normalized (Norm.) percentage of Max. to indicate what percentage of initial system size that Max. change represents. For example, the Max. change of 176 for the HSQLDB represented an increase of 271% in that system over its original size (of 65 classes).

We also include the approximate variance (Var.) values for the set of changes for versions of each system. For example, the variance of the set of changes from version to version of the HSQLDB system was 15336.

Table 1. Summary change data for the seven systems (all versions)

System	Max Ch	Norm.	Min Ch	Var.	Med Ch	Mean Ch
HSQLDB	176	271%	0	15336	23.5	58.6
Jasper Reports	183	22%	-77	11696	13.5	23.3
EasyWay	16	9%	-18	190	0	0.76
SwingWT	160	320%	0	39327	20.5	27.19
JAG	3	2%	-12	17	1.0	1.0
JBoss	4537	115%	-4506	5073056	245	476.9
Tyrant	103	84%	-85	1657	0	3.58

From Table 1 we see considerable variation in the behaviour of the systems. However, the mean change is always positive indicating a tendency to grow in size over time. This is most pronounced for JBoss. The size of a release or change is also most erratic for the JBoss according to its variance. The EasyWay, JAG and Tyrant systems all have relatively low median and mean change values.

We could consider a *stable* system as one with a close to zero mean change value and low variance. Although no single system satisfies these criteria JAG and EasyWay seem the most stable of our seven systems. Remarkable is the fact that Tyrant contained twenty-three 'transitions' from one version to the next, where no change in the number of classes was noted (and could be considered the most stable of the seven systems even though it does not have the smallest variance of the systems studied). It is also worth noting that the number of versions studied is not a particularly good indicator of size of change. One of the lowest mean changes belongs to the Tyrant system and the second largest mean change belongs to HSQLDB. If we view stability through the Norm. values from Table 1, then the JAG and EasyWay systems figure prominently again (as does the JasperReports system).

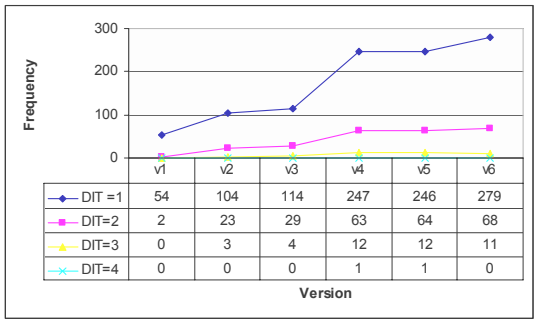
4. DATA ANALYSIS

Our analysis now considers the evidence to support our conjecture that the inheritance hierarchy grows in 'breadth' rather than 'depth'. We begin with a coarse-grained analysis of the trends in numbers (i.e. frequency) of classes at each DIT level on a version-by-version basis for each of the seven systems.

4.1 Coarse-grained DIT analysis

Figure 1 gives the frequency of DIT values for classes in the versions of HSQLDB and shows (apart from DIT level 4) a strong tendency for classes to be consistently added (i.e., representing a net increase) at DIT levels 1, 2 and 3 throughout. There is particularly strong evidence of classes being added at DIT levels 1 and 2 of the hierarchy. After version 3 however, the addition of classes to this system at both these levels starts to decline. There is only a single class at DIT level 4 and this class disappears by version 6. The strength of addition at DIT level 1 is illustrated by the fact that of the 302 classes added to this system over the course of the 6 versions, 225 were added to DIT level 1 and 66 added to DIT level 2. Combined, this represents 96.36% of the total. Only 11 classes were added to DIT level 3. Thus we have a system that is characterized by change at the shallow levels of the hierarchy.

Figure 2 shows the same breakdown of the frequency of DIT values for versions of JasperReports and shows a similar upward trend to that of Figure 1. It appears that, again, the majority of classes were added at levels 1 and 2. Interestingly, the number of classes at levels 4 and 5 (10 and 4, respectively) did not change throughout the entire set of 11 versions studied. Of the 280 net classes added to JasperReports, only 13 were added to DIT level 3. In contrast, 267 classes, representing 95.36% of the total were added to DIT levels 1 and 2.



	v1	v2	v3	v4	v5	v6
DIT =1	54	104	114	247	246	279
DIT=2	2	23	29	63	64	68
DIT=3	0	3	4	12	12	11
DIT=4	0	0	0	1	1	0

Figure 1. DIT frequencies all versions (HSQLDB)

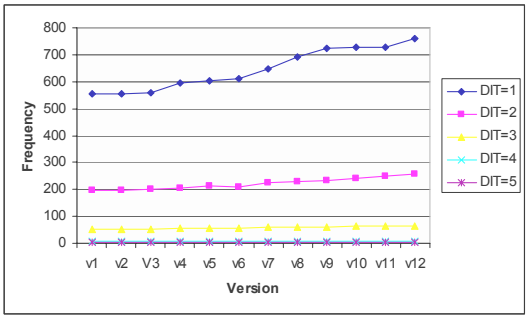


Figure 2. DIT Frequencies all versions (JasperReports)

Figure 3 shows the frequency of DIT values on an identical basis for the EasyWay system. The EasyWay system shows a different trend to that of HSQLDB and JasperReports. After version 2, there is a drop in the number of classes at DIT level 1 of the inheritance hierarchy and then the DIT fluctuates until version 8. It

then rises slowly until version 16, when the trend is then downwards again. Overall however, the net number of classes added at DIT levels 1 and 2 from a total of 14 classes added over all versions is 13 (i.e., 92.86%) of which 9 are at DIT level 1. It is noteworthy that, in keeping with the result for the JasperReports system, there is also very little activity at DIT levels 3 and 4 for System 3; only one class is added in total to level 3 throughout - zero classes were added for DIT level 4, which remained consistently at 1 throughout.

Figure 4 shows the DIT frequencies for the SwingWT system. A clear trend for classes to be added at DIT level 1 is evident again. In fact, for DIT levels 1 and 2, 400 and 83 classes were added, respectively. This compares with 19 added classes at DIT levels 3; a combined total of only 68 classes were added at levels 4, 5, 6 and 7. An interesting feature of levels 5, 6 and 7 is the fluctuation in the number of classes.

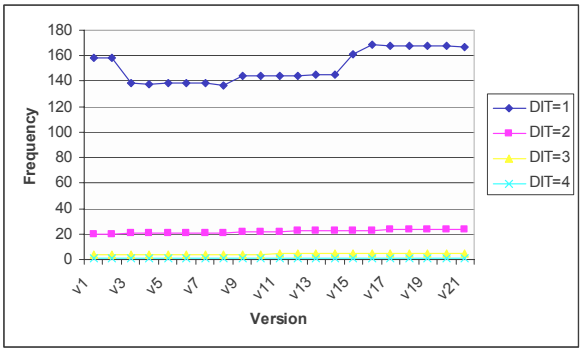


Figure 3. DIT frequencies all versions (EasyWay)

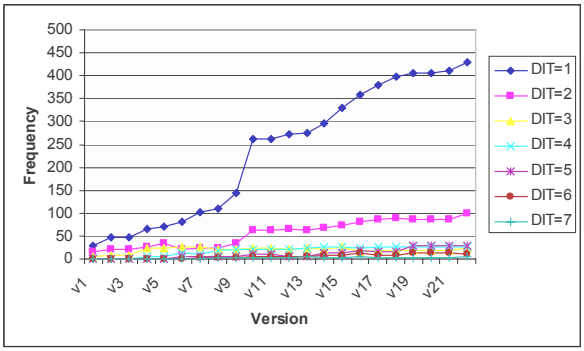


Figure 4. DIT frequencies all versions (SwingWT)

Figure 5 illustrates this feature; while fluctuating, the trend for classes at DIT level 5 (and to a certain extent level 6) is upwards.

Figure 6 shows the trend in DIT frequencies for the JAG system. In contrast to data from the other four systems (with the possible exception of the EasyWay system), the DIT level 1 values remain relatively static over the course of the versions studied. Only 2 classes are added to level 1 in total between versions 1 and 23. The number of classes at level 2 actually falls from 15 to 12 over the same number of versions. For DIT levels 3 and 4, in common with the JasperReports system, there is no change from their initial values.

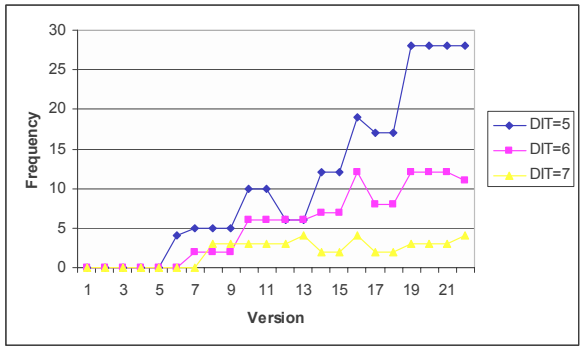


Figure 5. Classes at levels 5, 6 and 7 (SwingWT)

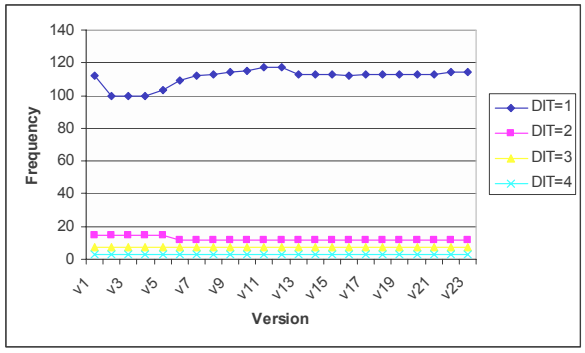


Figure 6. DIT Frequencies all versions (JAG)

For scaling purposes, Figure 7 shows the DIT level 1 trend for the JBoss (the system has the highest number of start and end set of classes). A fluctuating pattern can be seen and the sharp peak seems to occur between versions 20 and 23. Figure 8 shows the DIT frequencies for the remaining DIT levels 2-7. A striking feature of Figure 8 when compared with Figure 7 is the strong similarity between the graph for classes at DIT level 1 and those at DIT level 2. Both graphs peak and trough at the same times and there seems a common symmetry between the two lines. There is also a noticeable correspondence (although not nearly as pronounced) between the line graphs for DIT level 2 and DIT level 3. Both of these observations were also unexpected results from the analysis; they suggest that there is a strong correlation between the numbers of classes found at DIT level 1, DIT level 2 and, from the evidence presented, that at level 3).

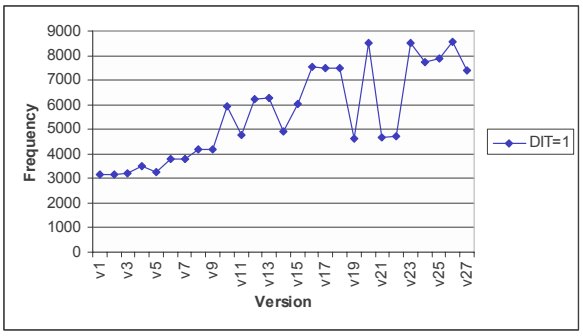


Figure 7. DIT level 1 frequencies for JBoss

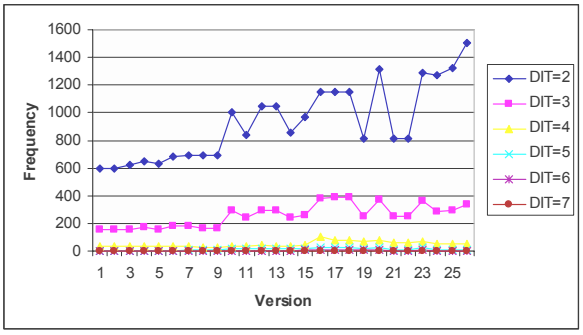


Figure 8. DIT frequencies for JBoss

Figure 9 shows the trend in DIT frequencies for classes in the Tyrant System. Version 5 seems to be the point where significant changes are made to the classes at each level and The rise in DIT level 1 and 3 values seems to be accompanied by a corresponding drop in DIT level 2 values. One noticeable feature of Figure 9 is the transition at version 26, when the number of classes at levels 1, 2 and 3 move from a 'plateau-like' pattern and start increasing.

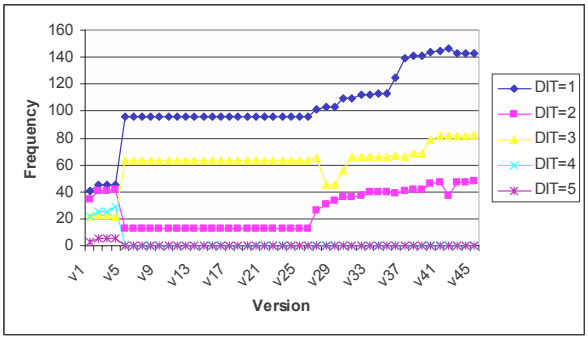


Figure 9. Frequencies for Tyrant

The emerging theme from Figures 1-9 is clear in terms of where the majority of classed are added. For each of the seven systems analyzed, DIT level 1 is where the main activity lies. To emphasize the difference between DIT levels 1, 2 and 3 we calculated that from a total number of 6397 net added classes over all versions of all systems:

- 5181 classes (80.99%) were added to DIT level 1,
- 972 classes (15.19%) were added to DIT level 2 and,
- 244 classes (3.81%) added to DIT level 3.

Moreover, only 25 classes were added to level 4 and 27 classes to level 5 (we note that only 4 of the 7 systems actually had classes at level 5). At deeper levels, there is strong evidence of classes being removed. At DIT level 5, 30 classes were added in total; at level 6, 11 classes were added and at DIT level 7, only 4 classes were added.

4.2 Specialization and Reuse Ratio

The main objective of the research in this paper was to show that the Java inheritance hierarchy tends to grow in width rather than depth. Based on previous studies [5, 14, 18], we believe that developers will add classes to low (shallow) levels of the inheritance hierarchy rather than extend existing classes. One measure that might further inform our analysis is the Specialization Ratio (SR) [19], which measures the extent of subclassing. A low SR implies that classes will tend to 'cluster' around lower levels of the inheritance hierarchy (i.e., DIT levels 1 and 2). A high specialization ratio suggests a high degree of subclassing. A further indication of the lack of subclassing is given by the Reuse Ratio (RR) [19]. An RR value close to 1 implies that the inheritance hierarchy is *narrow* and an RR value close to zero implies that the inheritance hierarchy is shallow [19]. Table 2 shows the summary data for the SR and RR metrics for the seven systems.

Table 2. SR and RR summary data for the seven systems

System	Med. SR	Max. SR	Med. RR	Max. RR
HSQLDB	0	0	0	0.8
JasperReports	0	0	0	0.86
EasyWay	0	0	0	0
SwingWT	0	14	0	0.75
JAG	0	0.33	0	0.86
JBoss	0	68	0	0.86
Tyrant	0	0	0	0.67

Table 2 gives a good representation of the lack of subclassing across the seven systems. The median SR and RR values are zero for all systems across all versions. Moreover, the maximum and standard deviation values

represent values from a very small sample of classes for which the SR and RR were computed. For example, for version 1 of the JBoss system, the SR values for only 8 of the 3934 classes were non-zero (i.e., 0.20%); equally, the RR for only 99 of the same 3934 classes was non-zero (i.e., 2.52%). For version 16, only 9 SR or RR values from the 5085 classes in that version were non-zero. For the 9082 classes in version 34, only 8 SR values and 118 RR values were non-zero. The same pattern applied to each of the other six systems. The very low values for the SR and RR values imply, by definition, that reuse through subclassing was very low in each of the seven systems and that the shape of the inheritance hierarchy very shallow. Considering the large number of classes added at DIT level 1 and 2 and documented in the preceding sections, this does not come as a surprise. However, this evidence does support the claim of the research that developers do not tend to add classes at deep levels of the inheritance hierarchy, but rather at shallow levels, itself causing a broadening of the entire hierarchy.

4.3 Number of children

A final indication of the structure of the inheritance hierarchy and how it may evolve is given by the Number of Children (NOC) metric. The metric measures the number of immediate subclasses for a class. To find support for our original claim, we would expect:

1. A relatively high proportion of the classes at DIT level 1 and 2 to have a large number of children.
2. Classes at DIT levels 3, 4, 5, 6 and 7 to have a very low proportion of children.

To investigate, we ranked all NOC values in descending order and determined the DIT values for the first 50 classes in the generated sequence; we did this for both the first and last versions of each system (N.b., the SwingWT system only contains 50 classes in its first version hence why we chose the number 50 as a sample size). The extracted profile is given in Table 3. For example, for the HSQLDB system, when we ranked the top fifty NOC classes, 48 of the classes inspected (96%) had a DIT of 1 and only 2 classes had a DIT level 2. It can be seen that the vast majority of the classes are taken from DIT level 1. In every case except for the first version of Tyrant, over 50% of the top 50 classes when ranked on NOC were drawn from DIT level 1. In over half of the cases, this percentage exceeds 70% and, in five cases, equals or exceeds 80%.

Table 3. Breakdown of DIT ranked on NOC for first and last versions

HSQLDB	**DIT=1**	**2**	**3**	**4**	**5**	**6**
First version	48 (96%)	2	0	0	0	0
Last version	39 (78%)	11	0	0	0	0
JasperReports						
First version	30 (60%	16	3	1	0	0
Last version	28 (56%)	18	3	1	0	0
EasyWay						
First version	43 (86%)	6	1	0	0	0
Last version	41 (82%)	8	1	0	0	0
SwingWT						
First version	29 (58%	16	5	0	0	0
Last version	28 (56%)	10	3	5	2	2
JAG						
First version	40 (80%)	5	5	0	0	0
Last version	40 (80%)	5	5	0	0	0
JBoss						
First version	37 (74%)	10	3	0	0	0
Last version	39 (78%)	9	2	0	0	0
Tyrant						
First version	16 (32%)	16	8	10	0	0
Last version	31 (62%)	11	8	0	0	0

Moreover, the top *ten* classes (ranked on NOC) were invariably drawn from DIT levels 1 and 2. For example, of the top ten classes for the first version of HSQLDB, 9 were at DIT level 1 and 1 at DIT level 2. Equally, for the first version of SwingWT, the top ten classes comprised 8 classes at DIT level 1 and 2 classes at level 2. For the final version of the JBoss system, 9 of the top ten classes were at DIT level 1 and only 1 at DIT level 2. Figure 10 shows this trend and the large number of children associated with those classes (NOC values actually ranged from 14 to 69); this breakdown is typical of the seven systems studied. Figure 11 shows the same data for the final version of Tyrant.

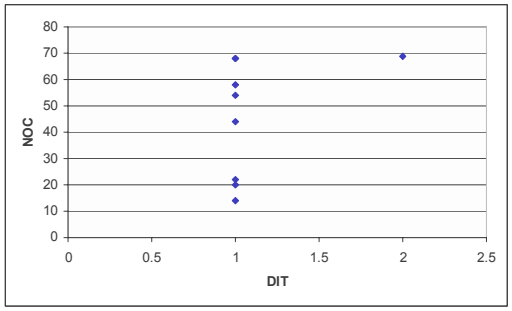


Figure 10. DIT and ranked NOC for JBoss

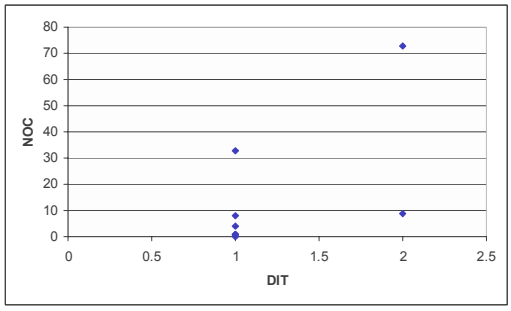


Figure 11. DIT and ranked NOC for Tyrant

The data in Table 3, and the evidence presented confirms our claim that the majority of activity is at DIT levels 1 and 2, with very little activity at, and beyond, level 3. Only 21 of the 700 classes (3.0%) from Table 4 were found to be at levels 5-7.

5. DISCUSSION

Many issues arise from the analysis in this paper. The population is non-trivial OSS projects that have undergone protracted maintenance. One important aspect that needs to be considered is the threat to the validity of the study. Firstly, we have to consider the extent to which our non-random sample has impacted our ability to generalize. However we have chosen a set of application domains ranging from computer games to a database application. Secondly, we have looked at different numbers of versions of each of the seven systems. While ideally, we would have liked to have had the same number of versions for each system, we wanted to extract as much information about available data as possible., Thirdly, while we can make observations about numbers of classes at different levels of the inheritance hierarchies, we can not say with any certainty, or quantify with any certainty, the movement of classes between the different levels. This would require a finer-grained analysis of the code and we leave that and the refactoring aspects for future work. Fourthly, since we restrict our analysis to structural aspects of the evolution we do not know *why* the developers made the choices that they have.

A question that arises from the study is whether we should consider the evolution of systems at shallow levels as bad practice, since it contradicts the original aim of inheritance? Our belief is maybe not. Developers will nearly always modify systems in the easiest and quickest way possible and from that perspective we could not really expect 'ideal' trends to occur. Furthermore, systems will inevitably deteriorate over time and re-engineering effort by developers is a luxury that cannot usually be afforded. In other words, it is not bad practice that leads to evolution at shallow levels, merely a 'fact of life' in the maintenance world that systems will evolve in a manner that conforms to forces dictated by the original architecture and by previous maintenance effort.

Many systems may not be amenable to deep inheritance hierarchies in the first place, so any additional classes will always be placed at shallow levels. Previous studies have suggested that graphical-based systems are the most amenable to extension through inheritance (interestingly, the SwingWT system in our study did exhibit high levels of inheritance up to DIT 7) [18].

One interesting aspect of OSSs is that the developers are often geographically and often time-zone separated from each other. Often the design documents are not available to each of the 'contributors'. We offer the explanation that for OSSs, developers may add classes at shallow levels of the inheritance hierarchy because they are unaware of the 'bigger design picture'. Of course, this does not explain why for previous studies where proprietary software was used, the same observations have been made, although scale might have a similar impact. In addition, an anecdotal claim of many developers is that the original designs of many proprietary systems are not updated as and when changes to the software are made and this renders those designs virtually unusable. The explanation for the lack of available design documentation in OSS may therefore be mirrored by outdated designs in proprietary software.

We also need to consider the implications of our study. One major implication of the effective *flattening* of the inheritance hierarchy is the potential maintenance headache of modifying a class with many children (i.e. its dependencies). Inheritance is a form of coupling [8] and, in this sense, a short-term 'easy fix' may be at the expense of long-term problems - refactoring may have a large role to play in this sphere of developer activity [12, 15]. Finally, it is interesting and ironic that there is previous empirical evidence to suggest that deep levels of inheritance have been blamed for the existence of faults; yet, we could suggest that by avoiding those deep levels of inheritance, the problem may simply have been devolved to shallower levels of inheritance (further empirical studies would be needed to support this claim).

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have described an empirical analysis of the trends in inheritance over multiple versions of Java OSSs. Previous studies have suggested that developers tend to avoid the use of inheritance at deep levels and that consequently, systems will evolve at very shallow levels (they will grow 'width-wise' rather than 'depth-wise'). The aim of the research described in this paper was to demonstrate whether or not this was the case. A tool (JHawk) was used to extract inheritance-based metrics from seven OSSs. The results in this paper confirm for

OSSs what many of the earlier reported studies did for proprietary systems (low DIT levels). There is also a strong tendency for classes to be added at levels 1 and 2 of the hierarchy rather than at deeper levels. Over 96% of classes added over the course of the versions of all systems were either at level 1 or level 2. This result was supported through analysis using the Specialization Ratio, Reuse Ratio and Number of Children metrics, which showed the extent of reuse in, the shallowness of, and width within, the inheritance hierarchy, respectively. These metrics supported and informed our analysis of the DIT and NOC metrics forming the main thrust of the paper.

The results have relevance for developers in terms of systems maintenance and refactoring. Predicting change-prone areas of systems will help to target refactoring effort and this may impact the localization of faults. If the majority of additions of classes are made at shallow levels of the hierarchy, then that is *possibly* where the faults will be likely to be found as a system evolves. This study also contributes to an empirical body of knowledge on inheritance and our understanding of software engineers' views on inheritance; we urge further empirical studies to refute or support our claims [21, 22]. Our future work will focus on two key areas. Firstly we want to investigate the trends in faults associated with the seven systems studied in this paper. Secondly, we want to investigate the potential for refactoring inheritance hierarchies [1, 15] and through measurement, studying the effect induced.

REFERENCES

[1] Advani, D., Hassoun, Y., and Counsell, S., Extracting Refactoring Trends from Open-source Software and a Possible Solution to the 'Related Refactoring' Conundrum. Proceedings of ACM Symposium on Applied Computing, Dijon (SAC 2006), France, April 2006, pages 1713-1720.

[2] Basili, V. R., Briand, L. C. and Melo, W. L., How Reuse Influences Productivity in Object-Oriented Systems, Communications of the ACM, 39(10), pp. 104-116, 1996.

[3] Basili, V., Briand, L. and Melo, W., A validation of object-oriented design metrics as quality indicators. IEEE transactions on Software Engineering, 22(10): 751-61, 1996.

[4] Basili, V. R., Viewing maintenance as reuse-oriented software development, IEEE Software, 7(1). pp. 19-25, 1990.

[5] Bieman, J. and Zhao, J., Reuse through inheritance: A quantitative study of C++ software, ACM Symposium on Software Reuse, Seattle, Washington, pp. 47-52, 1995.

[6] Booch, G. Object-Oriented Analysis and Design with Applications, 2nd ed., Benjamin/Cummings, 1994.

[7] Dinh-Trong, T., and Bieman, J., Open Source Software Development: A Case Study of FreeBSD, Proceedings of 10th IEEE International Symposium on Software Metrics, Chicago, USA, 2004, pages 96-105.

[8] Briand, L.C., Daly, J.W. and Wust, J.K. 1999, A unified framework for coupling measurement in object-oriented systems, IEEE Transactions on Software Engineering, vol. 25, no. 1, pp. 91-121.

[9] Capiluppi, A., Morisio, M., and Ramil, J., Structural Evolution of an Open Source System: A Case Study, Proceedings of the 12th International Workshop on Program Comprehension, Bari, Italy, pages 172-182, 2004.

[10] Cartwright, M., and Shepperd, M., An Empirical Investigation of an industrial object-oriented (OO) system. IEEE Transactions on Software Engineering, 26(8), pp. 786-796. 2000.

[11] Chidamber, S.R. & Kemerer, C.F., A metrics suite for object oriented design, IEEE Transactions on Software Engineering, vol 20, no.6. pp. 467-493, 1994.

[12] Counsell, S., Hassoun, Y., Johnson, R., Mannock, K., and Mendes, E., Trends in Java Code Changes: the Key to Identification of Refactorings? International Conference on the Principle and Practice of Programming in Java. Ireland, p 45-48, 2003

[13] Chidamber, S. R., Darcy, P. D., Kemerer, C. F., Managerial use of metrics for object-oriented software: an exploratory analysis, IEEE Transactions on Software Engineering, 24(8), pp. 629-639, 1998.

[14] Daly, J., Brooks, A, Miller, J., Roper, M. and Wood, M, Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software, Empirical Software Engineering: an International Journal, 1(2), pp. 109-132, 1996.

[15] Fowler, M. (1999) Refactoring: Improving the Design of Existing Code. New York: Pearson Education.

[16] Girba T., and S. Ducasse, S., Modeling History to Analyse Software Evolution, Journal of Software Maintenance and Evolution, 18(3), pages 207-236, 2006.

[17] Girba, T., Lanza, M., and Ducasse, S., Characterizing the Evolution of Class Hierarchies, Software Maintenance and Reengineering, 2005.CSMR 2005.Ninth European Conference on Software Maintenance and Reengineering, pp. 2-11.

[18] Harrison, R., Counsell, S., and Nithi, R., Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented Systems, Empirical Assessment in Software Engineering, Keele, April, 1998.

[19] Henderson-Sellers, B. (1995), Object-oriented metrics: measures of complexity, Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

[20] JHawk tool: (http://www.virtualmachinery.com/jhawkprod.html).

[21] Kemerer, C.F., and Slaughter, S., Need for more Longitudinal Studies of Software Maintenance, Report from the Proceedings International Workshop on Empirical Studies for Software Maintenance, Monterey, California., 1996, Empirical Software Engineering: An International Journal, 2(2), pages 109-118, 1999.

[22] Kemerer, C.F., and Slaughter, S., An Empirical Approach to Studying Software Evolution, IEEE Transactions on Software Engineering, 25(4), pages 493-509, 1999.

[23] Lehman, M., Ramil, J., Wernick, P., Perry, D., and Turski, W. M., Metrics and Laws of Software Evolution - The Nineties View, IEEE International Symposium on

Software Metrics (METRICS 97), Albequerque, USA, pages 20-32, 1997.
[24] Prechelt, L., Unger, B., Philippsen, M., and Tichy, W., A controlled experiment on inheritance depth as a cost factor for code maintenance, The Journal of Systems & Software, vol. 65, no. 2, pp. 115-126, 2003.
[25] Wood, M., Daly, J., Miller, J. and Roper, M., Multi-method research: An empirical investigation of object-oriented technology, The Journal of Systems & Software, 48(1), pp. 13-26, 1999.