Where the Bugs Are: A Quasi-Replication Study of the Effect of Inheritance Depth and Width in Java Systems

S. Counsell[1], Stephen Swift[1], and A. Tahir[2]

[1] Dept. of Computer Science, Brunel University, London, UK
[2] School of Fundamental Sciences, Massey University, Palmerston North, NZ

Abstract. The role of inheritance in the OO paradigm and its inherent complexity has caused conflicting results in the software engineering community. In a seminal empirical study, Basili et al., suggest that, based on a critique of the Chidamber and Kemerer OO metrics suite, a class located deeper in an inheritance hierarchy will introduce more bugs because it inherits a large number of definitions from its ancestors. Equally, classes with a large number of *children* (i.e., descendants) are difficult to modify and usually require more testing because the class potentially affects all of its children. In this paper, we use a large data set containing bug and inheritance data from eleven Java systems (seven open-source and four commercial) to explore these two research questions. We explore whether it is the case that a class deeper in the hierarchy is more buggy because of its deep position. Equally, we explore whether there is a positive relationship between the number of children and bugs, if classes with large numbers of children are indeed more difficult to modify. Results showed no specific trend for classes deeper in the hierarchy to be more buggy *vis-a-vis* shallower classes; the four commercial systems actually showed a negative relationship. The majority of classes across the hierarchy were also found to have no children and those classes included the most buggy.

1 Introduction

The concept of inheritance is a cornerstone of the OO paradigm and plays a key role in the functioning of any reasonably-sized OO system [12]. Inheritance promotes reuse, encourages specialisation and is meant to reflect the way that humans naturally structure information [4]. Controversy still surrounds inheritance, not least because the deep levels that were typically envisaged in systems have not materialised; systems still tend to be relatively flat with shallow inheritance structures [5]. Past studies have also argued about the optimum level of inheritance, some suggesting that three levels of inheritance is the most efficient depth for developers to or that *flat* systems without any significant depth to the hierarchy is less likely to cause maintenance problems [7]. Very few empirical studies have looked at inheritance particularly with respect to "buggyness" in the past five to ten years and because of the different application type, nature,

artefacts, subjects used and research questions of studies that have looked at inheritance in the past and our desire to cast light on those results, the most we can hope to achieve is a "quasi-replication". So we see our work as supporting or refuting prior results, but with the many caveats of aforementioned factors.

In an early paper of Basili et al., [2] the six metrics of Chidamber and Kemerer (C&K) [6] were validated using eight C++ systems as a basis. The analysis included the Depth in the Inheritance Tree of a class (DIT) and Number of Children (NOC) C&K metrics. The DIT is a measure of the distance from the root (in Java, the root is class `Object` from which all classes inherit). So, a class at DIT level 1 is a class with only Object as its superclass; a class at DIT has 2 classes which it inherits from (in a line) between it and root etc. The NOC metric is the number of immediate descendants below a class. So, if two classes X and Y inherit from class Z, then Z has an NOC value of 2. The basis of their analysis and validation of the DIT metric was the assumption that "*....a class located deeper in a class inheritance lattice is supposed to be more fault-prone because the class inherits a large number of definitions from its ancestors.*" Equally, the study of NOC in the same paper was made on the basis that "*a class with numerous children has to provide services in a larger number of contexts and must be more flexible. We expect this to introduce more complexity into the class design.*" The assumptions of Basili et al., were heavily informed by the claims of the two metrics by C&K in their original paper.

In this paper, we use a large data set of seven open-source systems and four commercial systems containing thousands of classes to explore the relationship between DIT, NOC and bugs. Since we have no developer maintenance information for the systems analysed, we use bugs as a surrogate for maintenance complexity. We justify this on the basis that a class with higher numbers of bugs reflects a class which is likely to be complex and has, over its lifetime, been more difficult to maintain. We investigate two research questions. Firstly, we explore whether there is a correspondence between the level of inheritance and the incidence of bugs. Put another way, are classes at deep levels of inheritance more buggy than shallower classes? Secondly, is there a relationship between NOC and bugs? In other words, does a larger number of children belonging to a class (given by a higher NOC) indicate that the class will be more bug-prone? Results showed no specific trend for classes deeper in the hierarchy *vis-a-vis* shallow classes; the majority of the open-source systems showed no relationship between DIT and bugs. The four commercial systems showed a strong negative relationship for our first question. In terms of the other research question related to NOC, the vast bulk of classes across the hierarchy were found to have zero children, including the one hundred most bug-prone classes in every system. The message seems from our work is quite stark: empirical studies provide useful results and others may support and refute those results. Ultimately however, a "one size fits all" approach to the use of inheritance and advocating a specific depth or width of inheritance may simply be unattainable.

The remainder of the paper is structured as follows. In the next section, we describe preliminary information. We then analyse our two research questions

by examining bug data in the eleven systems (Sections 3&4). In Section 5, we look at related work and threats to our study before concluding and pointing to future work (Section 6).

2 Preliminaries

The data used in this study was originally produced by Madeyski et al., [11] and comprises a range of metrics from 43 releases of eleven Java open source and 27 releases of 6 industrial Java projects. The four industrial projects belonged to the insurance domain and all projects were developed by the same software development company. We note that the data used in this study is freely available to download from a repository link in the original paper by Madeyski et al., [11]. In contrast to our work, the analysis described in their paper was not related specifically to inheritance; the study empirically evaluated process metrics for those which most significantly improved defect prediction models based on product metrics. In their work, the cjkm tool [16] was used to collect the DIT and NOC metrics and the BugInfo tool, developed by one of the authors of [11], was used to collect bug information.

Table 1 shows the number of classes, the mean DIT and NOC and and corresponding median (med.) values for all classes where there was **at least one bug** across the eleven systems we studied[3]. We note that class `Object` is considered to be at DIT level 0. The four commercial systems are named Prop-1 to Prop-4 in the table. We can see, for example, that the Ant system has 350 buggy classes and the mean DIT of those classes is 2.55, with median 3. The mean NOC for this system is 0.67, with median 0. The DIT data seems to suggest that for the open-source systems, it is between DIT level 1 (below root) and DIT level 2 that the bulk of the bugs seem to lie; for the four commercial systems, there is a clear pattern for classes at DIT level 3 to be the source of problems - all four DIT mean and median values for the commercial systems are approximately 3. This is an interesting characteristic of the data since at least one study in the past has suggested that DIT level 3 may be the point beyond which code comprehension starts to become excessively complex for developers [7] and that is when problems start arising in the maintenance process. The four commercial systems stand out from the rest of the table in that sense.

A further striking feature of the table are the low values for NOC across all systems. For the set of bug-prone classes shown in the table, only one system (Camel) has an NOC value exceeding 1. The lowest NOC value was for the jEdit system (with an NOC value of just 0.20); all median NOC values were 0. The low values for NOC in our systems reflect the similar conclusion by Basili et al., [2] that most classes do not tend have more than one child and that flat systems (with low levels of DIT) are frequent [5].

[3] ant.apache.org, camel.apache.org, ant.apache.org/ivy, jedit.org, logging.apache.org/log4j, lucene.apache.org, poi.apache.org

Table 1. Summary of DIT and NOC (all systems)

System	#Classes	DIT mean	med.	NOC mean	med.
Ant	350	2.55	3.00	0.67	0.00
Camel	562	1.98	2.00	1.23	0.00
Ivy	119	1.73	1.00	0.52	0.00
jEdit	303	3.23	2.00	0.20	0.00
Log4j	260	1.71	1.00	0.32	0.00
Lucene	438	1.78	2.00	0.71	0.00
Poi	707	1.84	2.00	0.89	0.00
Prop-1	2436	3.02	3.00	0.79	0.00
Prop-2	1514	3.09	3.00	0.72	0.00
Prop-3	840	3.10	3.00	0.35	0.00
Prop-4	1299	3.45	3.00	0.80	0.00

3 DIT metric analysis

3.1 Summary of DIT data

For our analysis, we first explore the relationship between DIT and bugs and we then consider NOC. Henceforward, for expressiveness and clarity, we now refer to classes at inheritance level 1, 2 as simply DIT1, DIT2, respectively. Table 2 summarises the number of classes at each inheritance level (given by the DIT) for the eleven systems. Here, we report DIT6 and greater as a single total in the final column of the table for the purposes of brevity (this is chiefly because relatively few classes were found at levels greater than 6). For example, for the Ant system, there were 997 classes at DIT1, 498 classes at DIT2 and 521 classes at DIT3 etc.

The most noticeable feature of the table is the relatively stable numbers of classes evenly distributed across the four proprietary systems, when compared with the seven open-source systems. To put this into perspective, only 13943 classes from a total of 53649 (25.99%) for the four commercial systems were found at DIT1; for the open-source systems, the corresponding figure was 6917 from 13942 classes; this represents 49.61% of the total number of classes across the seven systems. For the four proprietary systems, DIT3 contained more classes than its corresponding DIT1 value in every case, reflecting the relatively even spread of classes in those systems. It is also interesting to note that the number of classes in the DIT$\geq$6 category for the seven open-source systems was far lower compared to the four commercial systems. Only jEdit shows significant numbers of classes at DIT6 and greater. jEdit is an editor tool and that type of system (based on panels, frames, boxes and labels) is acknowledged to contain a richer inheritance structure because of their very structured nature. We note that the maximum depths across all eleven systems was 9 (Prop-3) followed by DIT8 for jEdit, Prop-1 and Prop-4. Ant, Log4j and Prop-2 all had maximum DIT7, so the systems were broadly comparable in that sense.

Table 2. Summary of DIT levels (per system)

System/Depth	DIT1	DIT2	DIT3	DIT4	DIT5	DIT≥6
Ant	997	498	521	265	130	31
Camel	1827	683	584	182	127	25
Ivy	617	149	64	63	22	18
jEdit	1971	919	151	63	167	424
Log4j	349	130	43	18	6	11
Lucene	590	426	144	40	5	0
Poi	566	966	111	26	12	2
Prop-1	4343	1682	6364	4744	2087	3838
Prop-2	3100	1312	3939	2125	2150	527
Prop-3	2162	621	2724	1014	940	1406
Prop-4	1395	386	2129	2073	1969	619

Table 3 shows the eleven systems studied, the number of classes in each system, the number of bug-prone classes (i.e., classes containing at least one bug) and the number of bug-free classes of that total. It also provides the percentages that these values represent. For example, Ant comprised 2442 classes, of which 350 were bug-prone and 2092 bug-free. This represents 14.33% and 85.67% of the total, respectively. The table shows that the most buggy of the eleven systems was Log4j, where 46.68% of classes contained at least one bug. The least buggy system was jEdit, where only 8.2% of classes contained at least one bug. Generally speaking, the four proprietary systems were less bug-prone than the seven open-source systems; however Prop-4 stood out from the other three commercial systems, with a relatively high bug level (15.52%).

Table 3. System Summary by bugs

System	#Classes	Buggy	Bug-free	%Buggy	#Bugs
Ant	2442	350	2092	14.33	637
Camel	3428	562	2866	16.39	1371
Ivy	933	119	814	12.75	307
jEdit	3695	303	3392	8.20	943
Log4j	557	260	297	46.68	645
Lucene	1205	438	767	36.35	1314
Poi	1683	707	976	42.00	1377
Prop-1	23058	2436	20622	10.56	4102
Prop-2	13153	1514	11642	11.49	2167
Prop-3	8867	840	8027	9.47	1362
Prop-4	8571	1299	7272	15.52	1930

3.2 Correlation of DIT vs bugs

One way of determining the relationship between DIT and bugs is through correlation of the variables studied. Table 4 shows the results of correlation between DIT and bugs for the eleven systems and for completeness we provide three correlation coefficients: Pearson's r, Spearman's and Kendall's rank. Pearson's is a parametric measure and Spearman's and Kendall's coefficients are non-parametric, making no assumption about the data distribution [8]. Here, single asterisked values ("$*$") in the table represent significance at the 1% level and double asterisked values ("$**$") represent significance at the 5% level.

Table 4. Correlation of DIT and bugs

System	Pearson's	Spearman's	Kendall's
Ant	0.04	0.10	0.08
Camel	-0.04	-0.01	0.00
Ivy	-0.01	0.10	0.09
jEdit	-0.01	-0.02	-0.02
Log4j	0.10	0.20*	0.17*
Lucene	-0.04	0.00	0.00
Poi	-0.20*	-0.11*	-0.10*
Prop-1	-0.14*	-0.08*	-007*
Prop-2	-0.09*	-0.09*	-0.07*
Prop-3	-0.07**	0.04	0.03
Prop-4	-0.19*	-0.11*	-0.10*

The table shows a clear trend for the set of open-source systems; only two of the seven sets of correlation values show any significance and they are in opposing direction to each other (one is positive and one negative); for five of the open-source systems, there is clearly no notable relationship between DIT and bugs, with all values around the zero mark (i.e., just below or just above). This supports the view that there is no observable pattern to the distribution of bugs across the systems in terms of a DIT "landscape". So, it does *not* seem to be the case that classes at deep levels of the inheritance hierarchy are more buggy than at lower levels and, if we associate bugs with classes that are difficult to maintain, which is a reasonable assumption, then buggy classes do not seem to discriminate between one level or another.

The POI system stands out from Table 4, since the correlation values for this system are all negative and significant at the 1% level. In these cases, a higher DIT therefore suggests a lower incidence of bugs. In terms of OO theory, this is what we might expect to occur, since classes at deeper levels of the inheritance hierarchy would be smaller (because of specialisation), be more maintainable as a result and therefore be the source of fewer bugs. But that is not how in practice it seems to work out. For the set of four proprietary systems, a different, yet equally distinct pattern can be seen; for three of the four systems there is a

negative, significant association between DIT and bugs which was only present in one of the open-source systems (we saw the same for the POI system). The data for the four industry systems suggests that the deeper in the inheritance hierarchy a class resides, the lower its propensity for bugs. Prop-4 has the highest correlation coefficients overall.

From Table 4, we also see that the Log4j system is positively and significantly correlated at the 1% level. It is worth remarking that this system had the highest percentage of bugs (46.68%), as can be seen from Table 3. For this system, it appears that a higher DIT value does indicate a higher propensity for bugs, but this is probably because there are so many bugs in this system that this result was inevitable anyway. Table 5 shows the distribution of bugs across the DIT levels for this system. For example, at DIT1 there were 345 bugs, representing 52.83% of the total number of bugs (i.e., 636). The table also shows "bug-density" values which we define as the number of bugs at a particular DIT level, divided by the number of classes at that level containing at least one bug; this reflects the average number of bugs per class. If we now inspect these values, we see an interesting trend. The lowest bug-density of 2.25 was found at DIT1 and the highest at DIT4 (value 3.6). In other words, the highest propensity for bugs was found at DIT4 and the lowest bug density at DIT1.

Table 5. Bugs and bug-density (Log4j)

Depth	DIT1	DIT2	DIT3	DIT4	DIT5	DIT≥6
# Bugs	345	201	51	18	11	19
% Bugs	52.83	31.60	8.02	2.83	1.73	2.99
# Classes	153	71	18	5	4	8
Bug-density	2.25	2.83	2.83	3.6	2.75	2.38

Figure 1 shows the bug-densities for the seven open-source systems and Figure 2 the corresponding values for the four commercial systems. The most striking feature is for the jEdit system which stands out for the peak at DIT4 (bug density 9.43). The most notable feature across the two figures more generally is that the bug density varies, but for the four commercial systems that variance is relatively small. The bug density ranges between 1.14 and 2.44 for those systems, indicating that bugs do not seem to dominate any particular level. While the variance is wider for the open-source systems (values range between 0 and 3.67 in most cases) there seems to be no standout DIT. The two figures support thus the view that there is no harmful, useful or remarkable level of inheritance - a view stated unequivocally by Prechelt et al., [14].

3.3 The role of class size

One relevant question that arises from the preceding analysis is why there are such differences between the open-source systems and the proprietary systems in

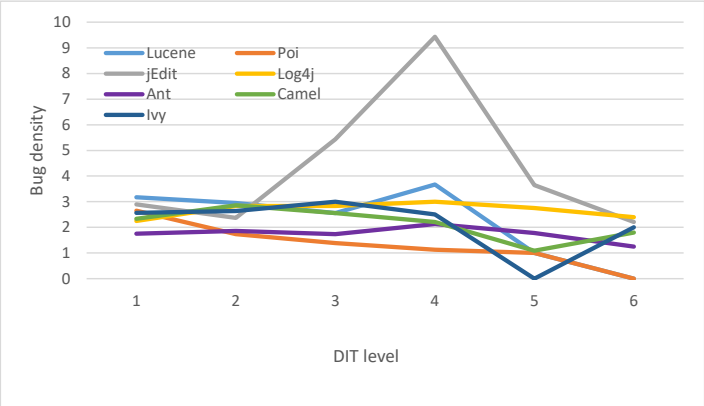

Fig. 1. DIT vs. Bug density (seven open-source systems)

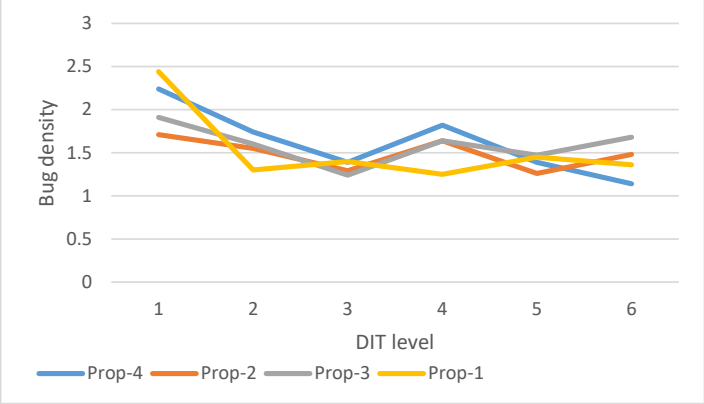

Fig. 2. DIT vs. Bug density (four commercial systems)

terms of results from Table 4 and from Figures 1 and 2? One possible explanation is that commercial systems are arguably better maintained and have cleaner code structures than open-source systems (although we cannot generalise in this sense). They may also be subjected more to practices such as refactoring [9] throughout their lifetime, which has the effect of stemming code decay and stopping code smells emerging. Smaller class sizes would be the norm and smaller classes we know are generally easier to maintain than larger classes, as well as generating fewer bugs on average. Table 8 shows the median and mean class sizes for each of the eleven systems. We again measure class size using the C&K WMC metric.

Table 6. WMC data per system

System	Median	Mean	Max.
Ant	8.74	5	120
Camel	6.93	4	166
Ivy	9.43	5	205
jEdit	7.59	3	413
Log4j	6.64	5	105
Lucene	7.68	5	166
Poi	12.23	9	134
Prop-1	4.84	4	347
Prop-2	4.42	3	140
Prop-3	4.30	3	136
Prop-4	3.96	2	212

The table shows that for only one open-source system (jEdit) is the median lower than the commercial systems and no mean WMC is lower in across the set of open-source systems. The most striking aspect of the table is the low class sizes for the four commercial systems. This evidence, together with the results from Table 4 and Figure 2 suggests that keeping classes relatively small may be one way of preserving a system's structure and, although our analysis is based on just eleven systems, potentially minimising the number of bugs in the system. To verify that smaller classes arise across the hierarchy, Table 7 shows the correlation values of DIT versus WMC for all eleven systems.

As we might expect, for the four commercial systems Prop-1 to Prop-4, Spearman's and Kendall's values are positive and significant at the 1% level. The same cannot necessarily be said of the seven open-source systems (where only Ivy shows the same type of relationship). Our belief that the commercial systems (Prop-1 to Prop-4) may be better maintained and looked after seems to have some traction. That said, whichever way this is looked at, bugs still do not seem to discriminate at any particular DIT level. It also suggests that there is no pattern in terms of the size of a class at any level.

Table 7. Correlation of DIT vs. WMC per system

System	Pearson's	Spearman's	Kendall's
Ant	0.04	0.09	0.07
Camel	-0.13*	-0.12*	-0.09*
Ivy	0.04	0.32*	0.26*
jEdit	-0.06	-0.08	-0.07
Log4j	-0.10	0.07	0.06
Lucene	-0.15*	-0.15*	-0.12*
Poi	-0.08**	0.06	0.05
Prop-1	-0.16*	0.26*	0.23*
Prop-2	0.01	0.24*	0.20*
Prop-3	-0.13*	0.29*	0.25*
Prop-4	-0.14*	0.34*	0.29*

Summary: No clear pattern to the relationship between the depth of a class (given by DIT) and the incidence of bugs was found for the eleven systems studied.

4 NOC metric analysis

The study of Basili et al., suggests from C&K's suite of metrics that classes with a high NOC value would be more complex and difficult to maintain. In their words: "*Classes with large number of children are difficult to modify and usually require more testing because the class potentially affects all of its children. Thus, a class with numerous children has to provide services in a larger number of contexts and must be more flexible.* We also believe that classes with a high NOC value will contain more bugs than classes with a low or zero NOC because of the extra complexity in classes with that high NOC. As per DIT, and for brevity, we now refer to a class with zero children as NOC0.

4.1 Correlation of NOC vs. bugs

To explore the relationship between NOC and bugs, we correlated their values across the eleven systems. Table 8 shows these correlation values for all classes containing at least one bug.

As we found for the DIT analysis, there is no clear trend in the data. For the open-source systems, there is only one system with a positive, significant relationship (Log4j). Interestingly, the same system showed the same result for DIT. This system had the highest percentage of bugs and it may simply be that is the only reason why the correlations were so significant. Most of the values in the table are close to zero, suggesting no obvious or standout relationship between NOC and bugs. For the four commercial systems, there is some evidence

Table 8. Correlation of NOC and bugs

System	Pearson's	Spearman's	Kendall's
Ant	-0.03	0.04	0.03
Camel	0.09**	0.06	0.06
Ivy	0.00	0.07	0.07
jEdit	-0.02	-0.14*	-0.13*
Log4j	0.13**	0.13**	0.11**
Lucene	0.00	0.05	0.04
Poi	0.00	-0.05	-0.05
Prop-1	0.04	0.10*	0.10*
Prop-2	-0.01	0.06**	0.06**
Prop-3	-0.01	0.02	0.02
Prop-4	0.00	0.02	0.02

of positive, significant relationships, but it only applies to two systems (Prop-1 and Prop-2).

One explanation for the lack of any trend in the NOC data and a feature of the eight systems studied by Basili et al., [2] is that most classes in the eleven systems had few or mostly zero children. Inspection of the NOC data revealed that for example, in the Ant system, 2163 of the 2442 classes (88.57%) had zero children; for the Camel system, the corresponding value was 90.15%. For Prop-1, the figure was 95.75% and for Prop-2, 94.45% of classes had zero children. We then listed the hundred most buggy classes in each of the eleven systems and found that for the Ant system, 77 of those 100 classes were at NOC0. For the Camel system, we found the corresponding value of 71 classes. For Prop-1, the number of classes with NOC0 in the top 100 buggy classes was 90 and for Prop-2, the figure was 88. Our original research question regarding whether classes with large numbers of children were more bug-prone seems to be largely overshadowed by the fact that so few classes in all the systems have children at all and that the most buggy classes are contained in that group.

Summary: No clear pattern to the relationship between NOC belonging to a class and the incidence of bugs was found for the eleven systems studied. The vast majority of classes had zero children.

5 Related work

There have been many (often conflicting) empirical results on the role of inheritance and as a community we are still no nearer establishing an optimal level of inheritance depth. Perhaps, as our study suggests, it will always elude us. Twenty-four years ago, Daly et al., [7] published the results of a controlled experiment into inheritance and its relationship with class maintainability using

the C++ language. The study evaluated subjects in their task of maintaining code written with different levels of inheritance (3 and 5); these were then compared with the effectiveness of similar tasks on systems containing no inheritance (flat systems). Results showed that subjects maintaining code with three levels of inheritance completed the tasks more quickly than those working on tasks on the flat system. The interesting observation however, was that subjects working on code with five levels of inheritance struggled with the inherent complexity at that depth and took longer to complete than tasks for the corresponding flat system. This suggests that beyond a level of inheritance, maintenance becomes problematic. One study that did "semi-replicate" the work of Daly et al., was by Prechelt et al. [14]. In their empirical study, they used a longer and more complex program and added a different type of maintenance task also. They cast doubt on the results of Daly et el., and concluded that: "....*previous results plus ours suggest that there is no such thing as usefulness or harmfulness of a certain inheritance depth as such*". Results from the paper herein seem to back up this claim.

In our paper, we also note that a big impediment to analysis of NOC was the high number of NOC0 values (i.e.. most classes having zero children). Interestingly, previous work on inheritance by Nasseri et al., [13] showed that, over time and as they evolved, inheritance hierarchies in open-source systems tended to collapse to bring classes up to shallower levels. Perhaps it is the case that as systems evolve, structures start to fragment through maintenance and it is simply easier to amalgamate classes and move them to shallower levels closer to the root than to try to maintain them at the deeper levels. This feature of systems ties with work by Bieman et al [3] who describe a study of nineteen C++ systems (with 2744 classes in total); only 37% of these systems had a median class DIT >1. Other studies have shown that flat systems (with low inheritance depths) are more easily maintained. Perhaps flat systems leads to fewer "mistakes" by developers and by implication, fewer bugs. Alternatively, moving classes to shallower levels is what the developers hope will happen. Finally, the danger of using inheritance were pointed out by Wood et al., [17]: inheritance should only be used with care and only when it is felt absolutely necessary.

5.1 Threats to validity

For any empirical study, need to consider the threats to its validity [15] [1]. Firstly, we only used eleven systems in our study. In this paper, our intention was to highlight key features of inheritance through the prism of DIT and NOC and while there is no such thing as the *right* number of systems to use in any empirical study, we feel that the work gives a fairly representative insight in the more broader issues typical of all systems. The four industrial systems were all developed in the same company and so we accept that this represents a "sub-threat" in this category. Secondly, we have made the assumption throughout the paper that bugs are a surrogate for complex classes and that a complex class will harbour and generate more bugs than a simple, less complex class. While

we accept that this may not always be true, we feel that in the absence of developer maintenance data to work with, this is a reasonable assumption to make. Thirdly, class size *per se* would influence the propensity for bugs (larger classes, more bugs) and we have looked at depth versus bugs as one indicator; however, this factor was not a key motivator of our work - rather that a class had experienced at least one bug. Fourthly, many of the previous empirical studies of inheritance used C++ systems, whereas we use Java. We defend this stance on the basis that while there are significant differences between the two languages, the OO paradigm is common to both and developer behaviour when maintaining systems does not seem to differ that greatly in the OO paradigm; one could argue that the different constructs used by OO languages may however have made a difference (this is a topic for future work). Fifthly, we have focused on classes where at least one bug was found and also of the twenty-five systems at their latest version point; this is because we wanted to understand the distribution of those bugs across inheritance as it stands presently. The study could be criticised because it failed to compare those results to classes without any bugs or indeed to look at version data. In our defence however, we were trying to quasi-replicate earlier work of studies where bugs were the dependent variable (and versions/version history were not explored). Sixthly, we have used a data set with different projects (open source and commercial), built by different developers, facing different development issues. This presents a risk to the generalisability of the results. Finally, the reader will have noted that the literature on empirical studies of inheritance has been fairly static over the past ten years. (The references in this paper are mostly from the mid-90's to around latest 2010.) We feel however, that this in no way undermines the need for studies like ours. In fact, it begs the question "why have there been no contemporary studies of inheritance on an empirical basis?" And also "what has changed in the past ten years?" such that no researchers are exploring this facet of systems any more.

6 Conclusions and further work

In this paper, we explored two research questions related to inheritance. The first explored the relationship between the depth of a class in the inheritance hierarchy and bugs and the second that a high number of children belonging to a class would render that class as more buggy. We found no evidence that classes at a specific depth of inheritance were more bug-prone than at any other depth. We did note some interesting differences between commercial and open-source systems, however, suggesting that the former are better maintained and looked after more generally. We also found no evidence to support the view that classes with a high NOC were any more buggy than other classes. The overwhelming number of classes had no children. Inheritance hierarchies, either through design or evolution do not tend to follow that pattern. One conclusion is that how a system evolves depends on factors such as the type of system, whether open-source or commercial and possibly system age. Work by Harrison et al., [10] suggested that large systems were equally difficult to maintain regardless of use

of inheritance. Perhaps it is the case that as systems grow and evolve, inheritance is just one more problem amongst an array of other problems that a developer faces. It thus becomes relatively less of a problem.

Future work will focus on extending the study to more commercial and open-source systems. We would also like to investigate the role that refactoring and code smells [9] play in the removal and possible introduction of bugs into code at different levels and the difference that such practices make to the shape of a system.

References

1. A. Ampatzoglou, S. Bibi, P. Avgeriou, and A. Chatzigeorgiou. Guidelines for managing threats to validity of secondary studies in software engineering. In *Contemporary Empirical Methods in Soft. Eng.*, pages 415–441. Springer, 2020.
2. V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Eng.*, 22(10):751–761, 1996.
3. J. M. Bieman and J. Zhao. Reuse through inheritance: A quantitative study of C++ software. In *ACM SIGSOFT Symposium on Soft. Reusability, 1995, Seattle, USA*, pages 47–52, 1995.
4. G. Booch. Object-oriented development. *IEEE Trans. Software Eng.*, 12(2):211–221, 1986.
5. M. Cartwright and M. Shepperd. An empirical investigation of an object-oriented software system. *IEEE Trans. Soft. Eng.*, 26(8):786–796, 2000.
6. S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.
7. J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering*, 1(2):109–132, 1996.
8. A. Field. *Discovering Statistics Using IBM SPSS Statistics.* Sage Publications Ltd., 4th edition, 2013.
9. M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.
10. R. Harrison, S. Counsell, and R. Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *J. Syst. Softw.*, 52(2-3):173–179, 2000.
11. L. Madeyski and M. Jureczko. Which process metrics can significantly improve defect prediction models? an empirical study. *Softw. Qual. J.*, 23(3):393–422, 2015.
12. B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1997.
13. E. Nasseri and M. J. Shepperd. Class movement and re-location: An empirical study of java inheritance evolution. *J. Syst. Softw.*, 83(2):303–315, 2010.
14. L. Prechelt, B. Unger, M. Philippsen, and W. Tichy. A controlled experiment on inheritance depth as a cost factor for code maintenance. *J. Syst. Softw.*, 65(2):115–126, 2003.
15. P. Runeson, M. Host, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering: Guidelines and Examples.* Wiley Publishing, 1st edition, 2012.
16. D. Spinellis. Tool writing: A forgotten art? *IEEE Softw.*, 22(4):9–11, 2005.
17. M. Wood, J. Daly, J. Miller, and M. Roper. Multi-method research: An empirical investigation of object-oriented technology. *J. Syst. Softw.*, 48(1):13–26, 1999.