Journal of Computing and Information Technology - CIT 18, 2010, 2, 159–165
doi:10.2498/cit.1001812

Java Method Calls in the Hierarchy – Uncovering Yet another Inheritance Foible

E. Nasseri and S. Counsell

School of information Systems, Computing and Mathematics, Brunel University, Uxbridge, United Kingdom

This paper describes an empirical investigation into method calls between classes at each level of the inheritance hierarchy in four, Java open source systems. We distinguish between method calls made to super classes in the hierarchy and external method calls made outside the line of super classes to the root. The premise on which the research rests is that classes should predominantly make use of super class functionality (as theory suggests) and relatively infrequent use of functionality outside those super classes. Results revealed that the most method calls were made to the methods of the classes where the majority of functionality resided (at shallow hierarchical levels) and not necessarily to the super classes of a class. The evidence presented therefore implies that developers are not using inheritance in the spirit originally intended and lends weight to the growing belief that OO inheritance has more practical foibles than theoretically stated advantages.

Keywords: method calls, inheritance, OO, empirical

1. Introduction

Exploring the evolution of inheritance hierarchy in a system can provide valuable insights into a system's dynamics from an inheritance perspective. Indeed, previous studies of inheritance have provided evidence of the ineffectiveness of inheritance beyond a certain level (Basili et al. 1996; Bieman and Zhao 1995; Cartwright and Shepperd 2000; Chidamber et al. 1999; Daly et al. 1996). In this paper, we investigate inheritance from a method call perspective and its impact in four, Java open source systems. The main research question that we explore is: *to what extent do classes take advantage of super class functionality offered* (*i.e., the subclass-super class relationships inherent in every inheritance hierarchy*)*when invoking the functionality of other classes?* The main motivation for the study in this paper arises from the dearth of empirical studies into how classes within an inheritance hierarchy interact and how that interaction might evolve as a system itself evolves (Capiluppi 2004; Lehman 1978; Lehman 1997). While we can view system evolution at a class level relatively easily (and view systems themselves as a collection of connected black boxes), any such study hides the lower-level granularity of functionality, activity and extent of coupling (Briand et al. 1999) that classes have across a hierarchy rather than up and down it. Results from our analysis of the four systems demonstrated that most method calls were made to the methods of the class where the functionality resided and *not* necessarily, as we might expect from a hierarchical inheritance structure, to the super classes of a class.

The work in this paper extends and builds on earlier work by the same authors where the evolution of class fields for the same four systems was examined (Nasseri and Counsell 2009; Nasseri et al. 2008). In Nasseri and Counsell (Nasseri and Counsell 2009), inconsistent and irregular patterns of field additions were noted as systems evolved, suggesting that field usage (i.e., their addition, deletion and movement) played a crucial role is OO system evolution; we extend that theme in this paper. The remainder of the paper is organized as follows. Section 2 describes the design of the empirical study including characteristics of the four systems, data collected and methodology used for the study. Section 3 presents data analysis on a system-by-system basis. In Section 4, we present relevant

literature that also casts doubt on the benefits of inheritance before we conclude and point to further work (Section 5).

2. Empirical Study Design

2.1. The Four Open-source Systems

For our study, we used a set of four, Java open-source systems chosen and downloaded from the systems available at `www.sourceforge.net`. The systems in ascending order of number of versions were as follows.

a) HSQLDB: a relational database engine implemented in Java. This system comprised 6 versions. HSQLDB consisted of 56 classes and 972 methods in version 1 and 358 classes and 3827 methods in version 6 (latest version).

b) JasperReports: a business intelligence and reporting system. This system comprised 12 versions. JasperReports contained 818 classes and 8201 methods in version 1 and 1098 classes and 10736 methods in version 12 (latest version).

c) SwingWT: an implementation of the Java Swing and AWT APIs. This system contained 22 versions. SwingWT contained 50 classes and 378 methods in version 1 and 620 classes and 6956 methods in version 22 (latest version).

d) Tyrant: a graphical fantasy adventure game. 45 versions of this system were studied. Tyrant contained 122 classes and 982 methods in version 1 and ended with 273 classes and 2374 methods in version 45 (latest version).

2.2. Data Collected

We used the JHawk tool (JHawk) to extract the following three OO metrics (Lorenz and Kidd 1994) and refer to the figure in Appendix A where appropriate to aid our explanation.

1) Depth in the Inheritance Tree of a class (DIT): this metric measures the number of ancestors of a class including class Object (Chidamber and Kemerer 1994). The DIT value for a class inheriting from only Object is assumed to be one. Henceforward, we refer to a class at a relatively *shallow* DIT level to mean one that is close to the root (i.e., class Object); equally, a relatively *deep* DIT level refers to a class further away from the root.

2) Calls to methods in the hierarchy (HIER): the HIER metric measures the number of method calls made to any super class in the hierarchy (JHawk). In the figure in Appendix A, MethodY in ClassB calls MethodX defined in its super class (ClassA) and is thus a contributor to the HIER metric.

3) Number of External methods calls (EXT): the EXT metric measures the number of method calls in a class to methods of other classes (JHawk) and *excludes* HIER calls. In the figure in Appendix A, MethodY in ClassB calls MethodZ in ClassC not in the same class hierarchy.

The basis of the research presented in this paper is to determine the extent to which classes in an inheritance hierarchy use functionality in their super classes (HIER) rather than that outside the line of classes to the root. OO guidelines would suggest that classes should predominantly use the specialised functionality of the super class-subclass relationship, not relationships between classes across the hierarchy.

3. Version Analysis

To investigate the research question, we explored the specific features of each system in the *latest* available version to establish how the values of EXT and HIER compared. We chose the latest version in each case, since differences between these two metrics were likely to be more pronounced as a system aged (and invariably decayed too). We consider each of the four systems in turn.

3.1. HSQLDB

Table 1 shows the numerical values of HIER and EXT at each DIT level for HSQLDB. The table format (after column 1 and *for each DIT level*) is as follows:

(1)Frequency and percentage of classes.

(2)Frequency and percentage of HIER (HIER).

(3) Frequency and percentage of EXT (EXT).

(4) Average number of HIER at each DIT level (HIER value divided by number of classes).

(5) Average number of EXT at each DIT level (EXT value divided by the number of classes).

DIT	Classes	HIER	EXT	Ave. HIER	Ave. EXT
1	279 (77.93%)	0	7050 (74.52%)	0	25.27
2	68 (18.99%)	104 (91.23%)	2012 (21.27%)	1.53	29.59
3	11 (3.07%)	10 (8.77%)	399 (4.22%)	0.92	36.27

Table 1. HIER, EXT and DIT (HSQLDB).

From Table 1, the majority of HIER (91.23%) exists at DIT level 2. The average HIER at DIT 2 is also greater than that of DIT 3; the number of EXT at DIT 1 is significantly higher than that of DIT 2 and 3. The average EXT at DIT 3, on the other hand, is significantly higher than that of EXT at DIT 1 and 2. We would expect classes at shallower levels of the hierarchy (i.e., DIT 1) to be more coupled to classes outside the hierarchy than classes at deeper levels since they 'inherit' relatively less potential functionality than classes at those deeper levels; unlike the former, the latter are able to take advantage of the functionality offered by their super classes.

Clearly, the HSQLDB does not conform to the model of how we would expect classes to behave and neither too our research premise.

3.2. JasperReports

Table 2 shows the same numerical values for the final version of JasperReports in the same format of Table 1. The maximum number of HIER (60.52%) exists at DIT 2 and, to a less extent, DIT 3. However, on average, the highest number of HIER (HIER/Classes) exists at DIT 3. We also see that DIT 4 and 5 contain zero HIER. The majority of EXT (56.92%) is found at DIT 1. (DIT 4 is where the minimum number of EXT is found.)

DIT	Classes	HIER	EXT	Ave. HIER	Ave. EXT
1	761 (69.31%)	0	10383 (56.92%)	0	13.64
2	258 (23.50%)	305 (60.52%)	6468 (35.46%)	1.18	25.07
3	65 (5.92%)	199 (39.48%)	1334 (7.31%)	3.06	20.52
4	10 (0.91%)	0	5 (0.003%)	0	0.5
5	4 (0.36%)	0	51 (0.28%)	0	12.75

Table 2. HIER, EXT and DIT (JasperReports).

The average EXT however, shows a different trend. On average, classes at DIT 2 and 3, respectively contain the highest EXT. In common with the HSQLDB system, we see a pattern of classes that should use functionality offered by super classes opting to use that of external (to the hierarchy) classes.

3.3. SwingWT

DIT	Classes	HIER	EXT	Ave. HIER	Ave. EXT
1	429 (69.19%)	0	2096 (34.59%)	0	4.89
2	99 (15.97%)	85 (24.50%)	956 (15.78%)	0.87	9.66
3	24 (3.87%)	22 (6.34%)	304 (5.02%)	0.92	12.67
4	25 (4.03%)	34 (9.80%)	890 (14.69%)	1.36	35.6
5	28 (4.52%)	112 (32.28%)	1289 (21.27%)	4	46.04
6	11 (1.77%)	85 (24.50%)	455 (7.51%)	7.73	41.36
7	4 (0.65%)	9 (2.59%)	70 (1.16%)	2.25	17.5

Table 3. HIER, EXT and DIT (SwingWT).

Table 3 shows the values of HIER and EXT for the final version of the SwingWT system. Despite the fact that the majority of classes reside at shallower DIT levels, the maximum number of HIER is from classes at DIT 5. It is notable that 28 classes (4.52% of all classes) account for 112 (32.28%) of HIER and 1289 (21.27%) of EXT in the system.

We also see that the majority of EXT is found at DIT 1; however, when taking into account the number of classes, DIT 5, 6 and 4, respectively are where the majority of EXT exists. On average, classes at deeper DIT levels have higher coupling, given by HIER and EXT, than classes at shallower levels. While we might have expected the majority of EXT values to occur at shallower levels of the hierarchy from a practical perspective (i.e., DIT 1 and 2), the evidence from the three systems examined so far suggests that this is not always the case. It is worrying that there is such a relatively high level of communication *across* the inheritance hierarchy. From a maintenance perspective, having to comprehend large disparate sections of classes dotted around the hierarchy is likely to be more time-consuming, error-prone and the cause of further faults on the part of the developer (Arisholm and Briand 2006; Arisholm et al. 2007).

3.4. Tyrant

Table 4 shows the numerical values of HIER and EXT for the final version of Tyrant. From Table 4, the majority of HIER (55.13%) exists at DIT 2. The average HIER also exhibits a similar trend. We also see a strong tendency to higher average EXT at shallower DIT levels and the trend seems to be downwards as the DIT increases. It would seem that in the case of the Tyrant system, there is some evidence of classes behaving in accordance with theory.

DIT	Classes	HIER	EXT	Ave. HIER	Ave. EXT
1	142 (50.01%)	0	4321 (64.79%)	0	30.43
2	49 (17.95%)	43 (55.13%)	1167 (17.50%)	0.88	23.82
3	82 (30.04%)	35 (44.87%)	1181 (17.71%)	0.43	14.40

Table 4. HIER, EXT and DIT (Tyrant).

3.5. Coupling Profile

As a further guide to the composition of classes in the four systems, Table 5 shows how method calls are distributed in each of the final versions of the four systems. The column format of Table 5 (after column 1 for systems) is as follows:

1) Number and the percentage of classes containing both HIER and EXT (HIER&EXT).

2) Number and percentage of classes containing neither HIER nor EXT (NONE).

3) Number and percentage of classes containing only HIER.

4) Number and the percentage of classes containing only EXT.

From Table 5, we see that the majority of classes contain at least one EXT. It is interesting that from all four systems, we find zero classes containing *only* HIER. Also remarkable from Table 5 is the large number of classes (286) in SwingWT containing zero HIER *and* EXT. SwingWT is a GUI application which has a maximum of 7 levels of DIT. Scrutiny of the data revealed that 6407 method calls (both HIER and EXT) were spread across just 334 classes of the 620 in total in the latest version. On average, each class therefore made 19.18 method calls. In comparison to the remaining three systems (HSQLDB: 30.92, JasperReports: 22.75 and Tyrant 26.25). SwingWT contained the minimum number of method calls per class.

System	HIER&EXT	NONE	HIER	EXT
HSQLDB	55 (15.36%)	48 (13.41%)	0 (0%)	255 (71.23%)
JasperReports	85 (7.74%)	274 (24.95%)	0 (0%)	739 (67.30%)
SwingWT	115 (18.55%)	286 (46.13%)	0 (0%)	219 (35.32%)
Tyrant	31 (11.36%)	16 (5.86%)	0 (0%)	226 (82.78%)

Table 5. The HIER and EXT data in the four systems (latest versions).

From a coupling perspective, this may sound encouraging, but from a design perspective, method calls are not evenly spread across all parts of the system (46.13% of classes contained zero HIER and EXT). This latter result confirms previous findings (Nasseri and Counsell 2008; Nasseri and Counsell 2009) that Swing has been poorly built and consistently shows features indicating that the system is decaying; it may also have been significantly patched up.

In the next section, we provide evidence to support the view that inheritance is not as straightforward a concept as theory might suggest. We draw on past empirical studies to demonstrate that inheritance is used sparingly at deep levels and that maintenance of systems at those levels could be problematic. The study we have presented supports the general view that inheritance is not being used appropriately and there may be many motives and explanations for how and why coupling evolves as it does.

4. Supporting Evidence

An experiment in which subjects were timed performing maintenance tasks on OO systems of varying levels of inheritance was undertaken by Daly et al. (Daly et al. 1996). Systems with 3 levels of inheritance (DIT 3) were found 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. The same experiment was replicated by Harrison et al. (Harrison et al. 1998) who found that flat systems (i.e. containing no inheritance) were easier to modify than systems containing three or five levels of inheritance. (Larger systems were equally difficult to understand whether or not they contained inheritance.)

Prechelt et al. (Prechelt et al. 2003) found that it took longer to maintain a program with deeper levels of inheritance than a program containing fewer inheritance features. In the study presented, we found very little activity below DIT level 3 and other studies seem to provide insights into why that might be the case. Indeed, a study of 19 C++ systems by Bieman and Zhao (Bieman and Zhao 1995) found that only 37% of these systems had a median class inheritance depth greater than 1. Cartwright and Shepperd (Cartwright and Shepperd 2000) collected a subset of metrics from a large telecommunications system comprising 133,000 lines of C++ and found a positive correlation between the DIT and number of user-reported problems, casting doubt on the use of inheritance for fault prevention. Wood et al. (Wood et al. 1999) suggests that inheritance should be used *only* when necessary and obvious and, even then, with care. The results herein provide one reason why this might be the case. In Chidamber et al. (Chidamber et al. 1998), three industrial OO systems were empirically investigated, and, again, none showed significant use of inheritance. They also reported relatively little use of inheritance in the system they analyzed. Finally, Basili et al. (Basili et al. 1996) 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 (at a shallower level); this hypothesis was found to be supported with statistical significance. Far from aiding maintenance, use of inheritance had the opposite effect. We would go as far as to say that Java method calls and their measurement (taken together with the results herein) represent the uncovering of 'yet another inheritance foible'.

5. Conclusions and Future Work

In this paper, we have presented an empirical analysis of four, Java open-source systems from a method call perspective. We distinguished between number of method calls made within the hierarchy (HIER) and number of external method calls (EXT) in classes of the studied systems. The JHawk tool was used to extract the HIER and EXT from final versions of the same four systems. The question we posed was: *do classes take advantage of super class functionality offered* (*i.e., the subclass-super class relationships inherent in every inheritance hierarchy*) *when invoking the functionality of other classes and to what extent?* Evidence suggests that the majority of method calls (both HIER and EXT) are made to the methods of classes where the majority of functionality exists, irrespective of the position of classes within the

hierarchy. In other words, super class relationships and the use of super class functionality by sub classes is systematically undermined. Only one of the four systems showed signs of the opposite.

The results may be of interest to software developers/practitioners as to how classes in a system interact and how that might change as a system evolves. We view the patterns observed as quite worrying for effective maintenance. Future work will focus on the analysis of systems from a refactoring perspective to see if that has any influence on the coupling of a system as it evolves (Fowler 1999; Nasseri et al. 2008).

References

[1] E. ARISHOLM, L. C. BRIAND, Predicting fault-prone components in a Java legacy system. Presented in *Proceedings of ACM/IEEE International Symposium on Empirical Software Engineering*, (2006) Rio de Janeiro, Brazil.

[2] E. ARISHOLM, L.C. BRIAND, M.J. FUGLERUD, Data mining techniques for building fault-proneness models in telecom Java software. Simula Research Laboratory Reports, 2007.

[3] V. BASILI, L. C. BRIAND, W. MELO, A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10), 751–761, 1996.

[4] J. BIEMAN, J. ZHAO, Reuse through inheritance: A quantitative study of C++ software. Presented in *Proceedings of ACM Symposium on Software Reuse*, Seattle, USA, 1995.

[5] L. C. BRIAND, J. DALY, J. WUST, A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1), 91–121, 1999.

[6] A. CAPILUPPI, M. MORISIO, J. RAMIL, Structural evolution of an open source system: A case study. Presented in *Proceedings of the International Workshop on Program Comprehension*, Bari, Italy, 2004.

[7] M. CARTWRIGHT, M. SHEPPERD, An empirical investigation of an OO system. *IEEE Transactions on Software Engineering*, 26(8), 786–796, 2000.

[8] S. R. CHIDAMBER, S. DARCY, C. KEMERER, Managerial use of metrics for object-oriented software: an exploratory analysis. *IEEE Transactions on Software Engineering*, 24(8), 629–639, 1998.

[9] S. R. CHIDAMBER, C. F. KEMERER, A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6), 467–493, 1994.

[10] J. DALY, A. BROOKS, J. MILLER, M. ROPER, M. WOOD, Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering: an International Journal*, 1(2), 109–132, 1996.

[11] M. FOWLER, Refactoring: Improving the design of existing code. Addison-Wesley, NJ, USA, 1999.

[12] R. HARRISON, S. COUNSELL, R. NITHI, Experimental assessment of the effect of inheritance on the maintainability of OO systems. Presented in *Proceedings of Empirical Assessment in Software Engineering*, Keele, UK, 1998.

[13] JHAWK tool: (`http://www.virtualmachinery.com/jhawkprod.html`).

[14] M. LEHMAN, Programs, cities, students, limits to growth? In: *Programming Methodology*, (D. GRIES Ed.), pp. 42–62, Springer Verlag, 1978.

[15] M. LEHMAN, J. RAMIL, P. WERNICK, D. PERRY, W. TURSKI, Metrics and laws of software evolution – the nineties view. Presented in *Proceedings of IEEE International Symposium on Software Metrics*, Albequerque, USA, 1997.

[16] M. LORENZ, J. KIDD, Object-oriented Metrics. Prentice Hall New Jersey, 1994.

[17] E. NASSERI, S. COUNSELL, System evolution at the attribute level: an empirical study of three Java OSS and their refactorings. Presented in *Proceedings of 32nd International Conference on Information Technology Interfaces*, Cavtat, Croatia, 2009.

[18] E. NASSERI, S. COUNSELL, Inheritance, 'warnings' and potential refactorings: an empirical study. Presented in *Proceedings of the 3rd International Conference on Software Engineering Advances*, Sliema, Malta, 2008.

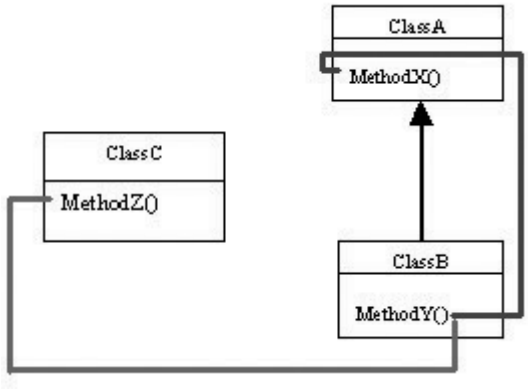
[19] E. NASSERI, S. COUNSELL, An empirical study of Java system evolution at the method level. Presented in *Proceedings of the IEEE International Conference on Software Engineering, Research, Management and Applications*, Hainan Island, China, 2009.

[20] E. NASSERI, S. COUNSELL, M. SHEPPERD, An empirical study of evolution of inheritance in Java OSS. Presented in *Proceedings of the Australian Software Engineering Conference*, Perth, Australia, 2008.

[21] L. PRECHELT, B. UNGER, M. PHILIPPSEN, W. TICHY, A controlled experiment on inheritance depth as a cost factor for code maintenance. *Journal of Systems and Software*, 65(2), 115–126, 2003.

[22] M. WOOD, J. DALY, J. MILLER, M. ROPER, Multi-method research: An empirical investigation of object-oriented technology. *Journal of Systems and Software*, 48(1), 13–26, 1999.

Appendix A: Method calls inside and across the inheritance hierarchy



Received: March, 2010
Accepted: April, 2010

Contact addresses:
Emal Nasseri
School of information Systems, Computing and Mathematics
Brunel University, Uxbridge
Middlesex, UB8 3PH, UK
e-mail: emal.nasseri@brunel.ac.uk

Steve Counsell
School of information Systems, Computing and Mathematics
Brunel University, Uxbridge
Middlesex, UB8 3PH, UK
e-mail: steve.counsell@brunel.ac.uk

EMAL NASSERI is currently a Research Associate at the University of Wolverhampton and interfaces with industry on a Knowledge Transfer Partnership. Emal received his PhD in 2009 from Brunel University exploring facets of Java inheritance hierarchies. His other research interests focus on the use of Java open-source software and the collection of data from these systems.

STEVE COUNSELL is currently a Senior Lecturer in the Department of Computing and Information Systems at Brunel University. His research interests are in software engineering and, in particular, the empirical study of software engineering facets such has refactoring and re-engineering. He also has a strong interest in software metrics and in data quality. He is also investigator on a two currently running funded software engineering research projects, both of which have strong industrial ties.