Class Movement and Re-location: an Empirical Study of Java Inheritance Evolution

E. Nasseri, S. Counsell and M. Shepperd

School of information Systems, Computing and Mathematics,
Brunel University, Uxbridge, Middlesex, UB8 3PH
{*emal.nasseri, steve.counsell, martin.shepperd*}*@brunel.ac.uk*

Abstract

Inheritance is a fundamental feature of the Object-Oriented (OO) paradigm. It is used to promote extensibility and reuse in OO systems. Understanding how systems evolve, and specifically, trends in the movement and re-location of classes in OO hierarchies can help us understand and predict future maintenance effort. In this paper, we explore *how* and *where* new classes were added as well as where existing classes were deleted or moved across inheritance hierarchies from multiple versions of four Java systems. We observed first, that in one of the studied systems the same set of classes was continuously moved across the inheritance hierarchy. Second, in the same system, the most frequent changes were restricted to just one sub-part of the overall system. Third, that a maximum of three levels may be a threshold when using inheritance in a system; beyond this level very little activity was observed, supporting earlier theories that, beyond three levels, complexity becomes overwhelming. We also found evidence of 'collapsing' hierarchies to bring classes up to shallower levels. Finally, we found that larger classes and highly coupled classes were more frequently moved than smaller and less coupled classes. Statistical evidence supported the view that larger classes and highly coupled classes were less cohesive than smaller classes and lowly coupled classes and were thus more suitable candidates for being moved (within an hierarchy).

Keywords: object-orientation (OO), maintenance, software evolution, inheritance, empirical analysis.

1. Introduction

Exploring system evolution can provide valuable insight into the location of changes, the size of changes made to, and the dynamics of, a system. The topic of software evolution has attracted the interest of researchers (Kemerer and Slaughter, 1999; Lehman, 1974; Lehman et al., 1997; Nasseri and Counsell, 2009; Nasseri et al., 2008). System decay and its consistent drain on resources is an ongoing problem for project managers and developers alike. This is as true for OO systems as it is for procedural.

In terms of the role of inheritance in systems, the message from research over the last ten to twelve years is that beyond a certain level of inheritance, problems with developer comprehensibility may arise. For example, a recent study by Nasseri et al., (2008) found that approximately 96% of incremental class changes over the course of the versions studied were at inheritance levels 1 and 2 (where level 1 is immediately below Object). Only 4% of changes were made at levels 3 and above. This same study has led to a number of further research questions and significant extensions to that previous study. While we can largely assume that a system will tend to grow as it evolves, interesting, yet largely unanswered questions, are how, when and why classes are moved, inserted and deleted across (and within) inheritance hierarchies as a system evolves? In other words, does the position of a class in an inheritance hierarchy change as a system evolves and if so where does it tend to be 're-located' and when? Additionally, if a class is moved, does its size and coupling lend itself to being moved, i.e. is it a relatively small and/or lowly coupled class? The aforementioned study by Nasseri et al., (2008) identified that the main activity of change in seven Java Open-Source Systems (OSS) occurred at levels 1 and 2. The unanswered question is therefore: what is the exact nature of the interplay between these two (and deeper levels) in terms of class relocation *as a system evolves* and are class size and cohesion factors?

In this paper, we manually inspected the position of a large sample of classes in the inheritance hierarchy of multiple versions of four Java OSS. The Depth of Inheritance Tree (DIT) inheritance metric of Chidamber and Kemerer (1994) was extracted from the systems using the JHawk tool (JHawk, 2008). The DIT is the inheritance depth of a class where the Java 'Object' class (and from which every class inherits) is at level zero; an immediate subclass therefore has DIT of 1 and the classes in the immediate level below that, a DIT of 2. We examined the position of each class in the inheritance hierarchy as each of the systems evolved. This allowed us to construct a pattern of the classes that tended to be moved, where they moved to, and when (in terms of version number). The work described in this paper thus extends our previous work to a finer-grained level; it does not simply look at overall patterns of increases or decreases in classes as a system evolves, but actually where the activity takes place. Thus the research question for this paper is: in the context of the tendency of systems to deteriorate in quality as they evolve *and* to exhibit shallow levels of inheritance depth, what observable, evolutionary patterns can be determined from class addition, deletion and movement around the inheritance hierarchies of systems?

We accept that the basis of this research question is a well-recognised problem in the software engineering community. Lehman's 'Law Two' of software evolution states that software complexity will increase as a system evolves unless work is done to prevent or reduce it (Lehman, 1974). This will inevitably lead to deterioration of the initial system design (e.g., through high coupling and low cohesion) as successive maintenance is applied to the system to cater for both shifting requirements and fault fixing. One way to impede structural complexity of a system is to restructure or refactor the system (Fowler, 1999). Class movement and relocation can thus play a significant part in reducing system complexity and aiding present and future software maintenance. A first step to answering the question posed, however, is to determine patterns in class movement on a large scale. Any understanding gained from this research will contribute to the very real problem of how to more effectively manage the evolution of software systems over time.

The remainder of the paper is organised as follows. In the next section, we present motivation for the research and related work. Section 3 presents the design of the study including a description of the systems used and the metrics collected and Section 4 presents data analysis on a system-by-system basis. In Section 5, we present the characteristics of moved and un-moved classes in the four systems. In Section 6 we present a discussion of the results before we present our conclusions in Section 7.

2. Motivation and related work

Our research is to highlight trends and features of an inheritance hierarchy as it evolves from the perspective of class re-location. Such a study can inform decision-making by a developer or project manager if those trends show any clear patterns. For example, observation of a certain subset of classes being consistently moved together (or duplicated) around the hierarchy suggests that the subset might need to be amalgamated for ease of reuse or simply refactored using the Collapse Hierarchy refactoring, where a subset of classes are merged (Fowler, 1999). Equally, if a single class containing relatively high levels of coupling is being moved frequently, it might need to be decomposed or a permanent location found for it. In addition, the analysis of class evolution makes it possible to identify the most change-prone parts in the systems and for remedial, re-engineering action to be taken as a result.

Many previous studies have analyzed inheritance in OO systems and most have questioned what is an appropriate level of inheritance. For example, the Depth of Inheritance Tree (DIT) metric, originally introduced by Chidamber and Kemerer (C&K) (1994) has been used extensively in many empirical studies (Cartwright and Shepperd, 2000; Daly et al., 1996; Harrison et al., 1998). Daly et al. reported an experiment in which subjects were timed performing maintenance tasks on OO systems with varying levels of inheritance. Systems with 3 levels of inheritance were shown to be easier to modify than a system containing no inheritance. Systems with 5 levels of inheritance were, however, found to take longest to modify. Harrison et al. (1998) replicated this experiment and reported that the systems with zero levels of inheritance were easier to modify than the systems with 3 *or* 5 levels of inheritance. They also found that large systems were equally difficult to maintain regardless of use of inheritance. Wood et al., (1999) suggested that inheritance should be used with care and only when needed. In two controlled experiments, Prechelt et al., (2003) reported that maintenance effort was positively associated with inheritance depth. In terms of further empirical studies into inheritance, Chidamber et al., (1998) empirically studied three commercial OO systems none of which showed significant use of inheritance. Bieman and Zhao (1995) described 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 (2000) described the collection of a subset of the C&K metrics from a large telecommunications subsystem (133,000 lines of C++). They reported relatively little use of inheritance in the system analyzed. However, when it did occur they found a positive correlation between DIT and number of user reported problems, casting doubt on the use of deep levels of inheritance. Basili et al., (1996) used the C&K metrics as predictors of fault-prone classes. Data from eight medium-sized management systems, developed in C++ were collected. Statistically significant results suggested that a class located deep in the inheritance hierarchy was more fault-prone than a class higher up in the hierarchy. This clearly implied that far from aiding maintenance, use of inheritance could have had the opposite effect.

Girba et al., (2005) in a study of two OSS systems, proposed measurements which summarized the evolution of source code over time. Using these measurements the authors defined rules to discover different traits of system evolution. Lehman et al., (1997) investigated the laws of software evolution originally introduced in Lehman (1974) using a financial transaction system as a basis. Results revealed that despite the long time gap (20 years) the 1970s approach to software metrics analysis was still applicable to software evolution. Capiluppi et al., (2004) in a study of the evolution of an OSS observed that the average size of files tended to stabilize as the system evolved. In their study, they used the number of folders, files and lines of code to quantify each version of the system. In a further study, Capiluppi and Ramil (2004) conducted an empirical analysis of two OSS (Arla and Mozilla) from an evolutionary perspective. They discovered some similarities in the evolutionary behaviour of the two systems at a higher level of abstraction and claimed that the growth rate of Arla was adaptable throughout its evolution while the evolution of Mozilla stagnated at certain points. The message from research into inheritance is that beyond a certain level of inheritance, problems with developer comprehensibility may arise.

Our study therefore informs the question that many of the results of these previous studies have posed; namely, that if there is a level of inheritance beyond which maintenance becomes problematic, then does a system 'naturally gravitate' towards that level through class re-location by developers? Crucially, can we suggest reasons at a low level of granularity as to why this may have happened (both questions unanswered by previous research)? While supporting the view that systems will tend to use initially (or through evolution) low levels of inheritance, the research presents an understanding of patterns and traits in that gravitation. Moreover, OSS present an increasing challenge to the software engineering research community and yet scrutiny of previous research in the area (i.e., that of empirical studies of such systems) reveals that, particularly with respect to inheritance, such studies are very limited. The study herein explores how evolutionary forces play a part in the structural evolution of OSS and whether there are specific characteristics of classes in those systems that make them candidates for re-location. The research provides a view of inheritance in OSS, in contrast to many earlier studies that tended to use proprietary software.

3. Study design

3.1 The four systems

A number of criteria were used to select the systems used in this study. First, they all had to be entirely Java systems. Otherwise this may present certain threats to the validity of such a study. The key threat is that developers might use the features of other languages to subvert the need for inheritance and therefore give us a false impression of evolutionary trends. This potential problem may be particularly acute in systems where user presentation and GUI is key. In other words, choice of a language like Visual Basic sub-system for handling such features would obviate the need for an extensive depth of inheritance in a Java sub-system. It may also be that some features of a programming language that Java is generally accepted as not particularly strong at handling (e.g., complex mathematical computation) might be 'outsourced' to different languages, weakening the value of the results. We justify our decision to select the systems from multiple application domains on the basis that it would enable us to generalize our findings into a broader OSS population. Second, the systems should be real, not contrived and sufficient versions should be available for a longitudinal study. Third the systems should consist of different sizes (in terms of number of classes). To accomplish this, systems were selected in 'number of downloads' order from sourceforge.net. In theory, relatively 'highly-used' systems are more likely to have received feedback on defects from users and to have evolved in terms of developer maintenance activity than less 'popular' systems as a result.

The process of selection resulted in four systems that were also the subject of previous empirical studies by ourselves (Nasseri and Counsell, 2009; Nasseri et al., 2008). Since the number of versions in some systems was very large so we chose to sample just the first, followed by every fifth and then final (major release) version given by sourceforge.net for each system.

1) HSQLDB: a relational database engine implemented in Java. This system comprised 6 versions. HSQLDB started with 56 classes in first version and comprised 358 classes by the final version. For this study we thus used versions 1, 5 and 6.
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. We thus used versions 1, 5, 10 and 12 for this study.
3) SwingWT: an implementation of the Java Swing and AWT APIs. This system comprised 22 versions. SwingWT started with 50 classes in its first version and increased in size to 620 classes by the final version. For this study we thus used versions 1, 5, 10, 15, 20 and 22.
4) Tyrant: a graphical fantasy adventure game. 45 versions of this system were studied. Tyrant started with 122 classes and ended with 273 classes by the final version. In the transition from version 4 to 5, significant change was observed in Tyrant, we therefore included versions 1, 4, 5, 10, 15, 20, 25, 30, 35, 40 and 45. The significant and extent of the changes in the Tyrant system at version 4 was a feature not shared by any of the other three systems and hence justifies our choice of its inclusion.

3.2 Data collected

For this study we used the JHawk tool to automatically collect the measures. JHawk was used in the two previous studies by the authors (Nasseri and Counsell, 2009; Nasseri et al., 2008) and has also been used by other researchers (e.g. Arisholm and Briand, 2006; Arisholm et al., 2007). The following data were collected:

- Depth of Inheritance Tree (DIT) metric of C&K. The DIT metric measures the total number of ancestor classes from a class to the root (level 1 of inheritance). Since every class inherits from Object, we consider DIT=1 as immediately below Object (i.e. we define Object to occur at DIT=0).
- Number of Methods (NOM) metric of Lorenz and Kidd (1994). The NOM measures the total number of methods in a class. This includes all methods (private, public and protected).
- Message Passing Coupling (MPC) metric of Li and Henry (1993). The MPC measures the total number of method calls in the methods of a class to methods of other classes. In other words, it measures the dependency of methods of a class to the methods of other classes.
- Lack of Cohesion Of Methods (LCOM) metric of Chidamber and Kemerer (1994). LCOM counts the number of method pairs accessing different fields/variables minus the number of method pairs accessing the same fields/variables.

The DIT metric was used to determine patterns in new, deleted and moved classes. The remaining three metrics were used to determine the characteristics of those classes. Having collected the metrics, we manually analyzed the position of each class in the inheritance hierarchies for each system according to the versions specified in Section 3.1. We were then able to categorise changes in the form of a) new classes b) deleted classes and c) moved classes in each of the inheritance hierarchies. For each class falling into one of these three categories, we then investigated whether class characteristics i.e., class size and coupling exhibited any particular or remarkable characteristics.

4. Data analysis

4.1 Summary Data

Table 1 presents summary statistics for incremental changes of classes in the four systems. By ‘change’ we mean either positive or negative ‘growth’ through the addition of, or deletion of, classes, respectively. The data are in the form of maximum (Max), minimum (Min), median (Med) and Mean change values in the number of classes across the all versions of the four systems. For maximum changes, Table 1 also indicates the normalized percentage of Max (Norm.) to indicate what percentage of initial system size that the Max change value represents. (This value gives a better indication of the extent to which a system underwent changes in a particular version, rather than explicitly looking at absolute values that convey less information if unrelated to system size.) For example, the Max change of 176 for HSQLDB represented an increase of 271% in that system over its original size (of 56 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.

From Table 1 the high Norm values for HSQLDB and SwingWT indicate that these systems changed significantly in size from their initial versions. This feature of the systems can also be seen in their Mean change values in the same table. SwingWT has the highest Var value of the four systems, suggesting that this system is subject to extreme fluctuations in the size of changes applied to it. Previous analysis of the Swing system (Counsell et al., 2007; Najjar et al., 2003) has shown that this system has a number of architectural problems, such that it is decaying and has been repeatedly “patched up” as a result. The low Norm. value for JasperReports suggests that the system does not change significantly from its initial version. The mean change and Var values for Tyrant are the lowest of the four systems. Tentatively, we consider Tyrant as the most *stable* system in the set of systems studied. It will be revealing to see if the pattern of movement of classes around the inheritance hierarchy for this system follows a similar trend of stability to support the values in Table 1; equally for the SwingWT system at the other end of the scale, (i.e. whether or not it experienced a wide fluctuation in class movement).

Overall, the change data indicate considerable variety between the systems but an overall tendency for them to grow in size. Table 1 shows that a) some systems tend to grow at a faster rate than others and this may be a direct reflection of their stability and b) significant changes can be in a downward (decreasing) as well upward (increasing) direction. In the next four sections, we present an analysis of each individual system.

4.2 HSQLDB

4.2.1. Summary detail. This system started with 56 classes of which 54 classes were at DIT=1 and only 2 classes at DIT level 2. The maximum DIT throughout all 6 versions of HSQLDB reached 4. Figure 1 shows the frequencies of number of classes at each DIT level in versions 1, 5 and 6. Also we see that DIT=1 is where the majority of classes were added. The number of classes at DIT=1 reached 279 in version 6. The number of classes at DIT=2 and 3 also increased but to a far lesser extent.

The values from Figure 1 show only the net increases or decreases in number of classes. On that basis, the following questions arise. Are all of the added class new? Have any classes been deleted? And most importantly for the thrust of the research in this paper, is there any movement of classes across inheritance hierarchy (or do classes tend to stay largely where they are)?

4.2.2. Class changes (versions 1-5). Table 2 shows the data for the movement of classes within the inheritance hierarchy between versions 1 and 5 in HSQLDB. Table 2 also shows the number of removed classes (RemC.) and the number of new classes (NewC.) added at each DIT level in the same transition. For example, 3 classes were moved from DIT=1 to DIT=2, 8 classes were removed from DIT =1 and 203 new classes added to this level. Evidence confirms the view that while most activity in terms of new classes seems to occur at DIT=1, there are certain occurrences of classes being pushed down the hierarchy (although usually in small numbers). We can only suggest that the developers moved classes from DIT=1 to DIT=2 so that those ‘moved’ classes could take advantage of functionality offered through inheritance by classes at DIT=1.

From visual inspection, one new class was added above one of the moved classes at DIT=1; the other two classes were added as subclasses of two existing classes at DIT=1. The total number of new classes in the system was 267 and only 8 classes were removed from DIT level 1; those eight classes were not re-located and were deleted from the system. It is possible, however, that the 203 added classes might have included those same 8 classes re-located but with a new name, simply amalgamated to form new classes or integrated into other classes. If any of these cases applied, then one suggestion is that these classes may have been the target of refactoring effort (Fowler, 1999), through use of renaming, decomposition of classes or collapsing of sub-hierarchies (evidence of which we found in other systems).

4.2.3. Class changes (versions 5-6). Table 3 shows the data for the movement of classes across inheritance hierarchy between versions 5 and 6 in HSQLDB, in exactly the same format as Table 2. In version 6, the total number of classes in the system increased from 323 to 358. One class was moved from DIT=1 to DIT=2 and 3 classes were moved from DIT=2 to DIT=1. One new class was added above the moved class from DIT=1 to DIT=2, suggesting that the addition of this new class could be part of an ‘Extract Superclass’ refactoring (Fowler, 1999). The 3 displaced classes from DIT=2 were separated from their respective superclasses and moved to DIT=1. There seems to be evidence of movement of classes from DIT=2 to DIT=1 from the data illustrated so far.

The maximum DIT in version 5 was 4 and only one class could be found at this level. In version 6, the same class was removed thereby reducing the maximum DIT in this version to 3. From Table 3, we also note that there is very little movement of classes within the inheritance hierarchy. The majority of changes are incremental (i.e., new classes) suggesting that for this system, a well structured inheritance hierarchy was in place and that lent itself well to the addition of classes. Figure 2 shows the frequencies of number of new classes (NewC), removed classes (RemC) and moved classes (MovC) in the studied versions of HSQLDB.

One further suggestion as to why there was so little activity within the inheritance hierarchy might be due to the nature of OSS development and the geographical dispersal of developers in many cases. Since the full set of documentation for an OSS system is not always available, we believe that OSS may often be maintained based on a model such as the *Iterative-Enhancement* (I-E) maintenance model of Basili (1990). That model is usually used for maintenance of proprietary systems when the full set of requirements is not fully understood by developers. The underlying principle of the I-E model is to re-design, reuse and/or replace parts of an existing system that is exhibiting features that render it difficult to maintain. Evidence presented so far suggests this might be an appropriate model for systems which evolve in a haphazard fashion.

4.3 JasperReports

4.3.1. Summary detail. This system started with 818 classes in version 1 and expanded to 1098 classes by the twelfth and final version studied. The maximum DIT for JasperReports remained at 5 throughout all 12 versions. Figure 3 shows the frequencies of number of classes at each DIT level in every fifth version and final version. The number of classes at DIT=1 has the highest growth rate. This trend was also observed in a previous study by the authors (Nasseri et al., 2008). The number of classes at DIT=2 and 3 also increased, but at a slower rate. Interestingly, the number of classes at DIT=4 and 5 stayed static at 10 and 5, respectively throughout.

Part of the JasperReports system thus showed no changes (level 4 and 5 of hierarchy) despite the fact that other parts of the system were undergoing frequent change. This points to the possibility that classes at deep levels are not usually the focus of developer attention and often ignored. However, the lack of classes at those levels may mean that they remain relatively untouched by the developer anyway.

4.3.2. Class changes (versions 1-5). Table 4 shows the evolution of classes across inheritance hierarchy between version 1 and 5 in the style of Tables 2 and 3. In the transition from version 1 to 5 of JasperReports, we see only 5 classes were moved within the inheritance hierarchy. Visual inspection revealed that 2 new classes were added at DIT=1; 3 classes from DIT=1 were moved as subclasses of one of the 2 new classes and one class from DIT=1 was moved as subclass of the second new class. Only 1 class was moved from DIT=1 to DIT=3. The maximum DIT of that particular hierarchy reached 3 as a result.

4.3.3. Class changes (versions 5-10). Table 5 shows the profile for JasperReports between versions 5 and 10. In the transition from version 5 to 10, only 2 classes were moved from DIT=1 to DIT=2. The majority of changes in the system occurred at higher levels of hierarchy (levels 1 and 2). Table 6 shows the profile for JasperReports between version 10 and 12.

From Table 6, 12 (i.e., 1.15%) of the total number of classes were moved from DIT=1 to DIT=2. From our inspection, we found that 3 new classes had been added at DIT=1. 7 existing classes from DIT=1 were moved as subclasses of just a single newly added class, 3 classes from DIT=1 and 1 class from DIT=3 were positioned as subclasses of the second newly added class. From the above analysis, we see that many of the changes essentially revolve around the 3 newly added classes. This localisation of change suggests that class movements might happen in clusters. Figure 4 shows the profile for changes in JasperReports. We see that the majority of changes are addition of new classes in JasperReports (total 296). The total number of removed classes was 16 and the total number of moved classes within the existing inheritance hierarchy was 21. Considering the large number of classes in JasperReports (818 in the first version and 1098 in the twelfth version) we believe inheritance hierarchy in the system was relatively stable (in terms of movement of classes within the hierarchy). We found no activity at DIT level 4 and 5 in the studied versions of JasperReports. This trend was also found for the same system in Nasseri and Counsell, (2009), where no activity in terms of number of *methods* was found at either level 4 or 5.

4.4 SwingWT

4.4.1. Summary detail. This system started with 50 classes and contained 620 classes by version 22. The maximum DIT for SwingWT reached 7. Figure 5 shows the frequencies of classes at DIT=1 to 3 and Figure 6 shows the same trend for DIT=4 to 7 in every fifth and final version of the system. For the early versions of SwingWT, the maximum DIT was 3 and that gradually grew to 7. Figure 5 again shows that the majority of classes were added where DIT=1.

4.4.2. Class changes (versions 1-5). Table 7 shows the movement of classes within the inheritance hierarchy between versions 1 and 5 of SwingWT. Only 1 class was moved up from DIT=3 to DIT=2. From our inspection, we observed that the two ancestor classes of that particular class were removed and 1 new class added above (this changed its DIT from 3 to 2). In the transition from version 1 to 5, we see that the system grew considerably. Overall, 83 classes were added and only 2 classes were removed.

4.4.3. Class changes (versions 5-10). Table 8 shows the profile for SwingWT in the transition from version 5 to 10. We see that between versions 5 and 10, 42 classes (i.e., 31.82% of all classes) were re-located across the hierarchy. Only 1 class was moved from DIT=1 to DIT=4. A new class was added above that class at DIT=1 (the name of the class was JSWMenuComponent and its new superclass AbstractButton); the same class and its new superclass were placed as subclasses of an existing class at DIT=2 (JComponent). Moreover, 10 classes were moved from DIT=2 to DIT=3. It was revealing that in version 5, those 10 classes were sibling classes of the JComponent class at DIT=2 which in version 10 were then placed as subclasses of JComponent. Three classes were moved from DIT=2 to DIT=4. Those three classes were subclasses of the Component class which were in turn moved as subclasses of the newly added class (AbstractButton) at DIT=3. Three classes were moved from DIT=2 to DIT=5. The 3 classes were subclasses of the JSWMenuComponent class at DIT=1 for which a superclass was added (AbstractButton) and was moved to DIT=4.

All movements of classes therefore revolved around only four classes (Component, at DIT=1, itself superclass of JComponent, AbstractButton and JSWMenuComponent). This suggests that the system was designed in such a way that the classes in one part of the system were highly amenable to easy movement which, in practice, could reflect a portable design. In addition, we found that some classes were moved, for instance, directly from DIT=1 to DIT=4; as a result, its subclasses were correspondingly moved from DIT=2 to DIT=5 hence, a dependency between groups of classes also seems to exist.

4.4.4. Class changes (versions 10-15). Table 9 shows the profile for SwingWT between versions 10 and 15. Between versions 10 and 15, 19 classes were moved within the inheritance hierarchy. A noticeable feature of Table 9 is the number of classes that moved from DIT=5 to DIT=1. Classes may move one or two levels as a result of addition/deletion of a class in the hierarchy however; movement of classes from *root* (below Object) closer to the *leaf* of hierarchy or vice versa implies a haphazardly structured hierarchy. Our analysis revealed that only one new class was added at DIT=3 and 2 classes were moved as subclasses of that class further reinforcing the view that 'dependent' classes do tend to move in clusters. Nine (47.37%) of the 19 moved classes were those classes which were moved within the hierarchy in the previous transition (between version 5 and 10). In other words, a large subset of the 19 moved classes between versions 5 and 10 were moved again between versions 10 and 15. This was an interesting result to emerge from our analysis, suggesting that a subset of classes were prone to movements. SwingWT is a GUI application which uses inheritance extensively. We believe the deep level of inheritance makes it harder for a developer to move classes within the hierarchy. Movement of one class may require several classes to be moved due to their superclass and subclass relationships which tend to be strongly tied. For example, if a class is moved from DIT=2 to DIT=4, any subclasses moved with that class changes its DIT from 3 to 5; we found ample evidence of this in SwingWT. In the transition from version 10 to 15, 32 classes were removed from the system, 19 (59.38%) of which were inner classes suggesting that inner classes are easily deleted from the system. This was also a revealing feature of the analysis. Inner classes might be easier to remove from a system because they are encapsulated within their outer enclosing classes. As such, inner classes have no dependencies (i.e. coupling) with other classes other than that imposed by the enclosing class.

4.4.5. Class changes (versions 15-20). Table 10 shows the number of classes moved around the inheritance hierarchy in the transition from version 15 and 20. Seventy classes (14.37% of the total) changed their position in the hierarchy. Only one class was moved from DIT=1 to DIT=2. One class was added at DIT=1 and the moved class was placed as a subclass of that new class. Similarly, one class was added at DIT=4 (in the most change-prone part of the system) and 8 classes moved to become subclasses of that new class. We also see that 12 classes were moved from DIT=3 to DIT=4. Those 12 classes were subclasses of a single class moved from DIT=2 to DIT=3 (its subclasses were moved from DIT=3 to DIT=4). Likewise, one of the 12 classes which were moved from DIT=3 to DIT=4 had 14 subclasses all of which were also moved from DIT=4 to DIT=5.

In the transition from version 15 and 20, we again found that the vast majority of movement of classes took place due primarily to the movement of their superclasses. Furthermore, we found that 14 (20%) of the 70 moved classes were those repositioned within the hierarchy in the previous transition (i.e., between versions 10 and 15); 38 (54.29%) of the 70 moved classes were those classes which were re-positioned in the transition between version 5 to 10. We also found the same 9 classes to be moved in every transition from version 5 to 20, supporting the view that there are certain subsets of classes so tightly coupled that they cannot be decomposed; they need to be moved around together (even though the functionality of *all* nine classes might not be required where they are moved). These classes would be ideal candidate classes for re-engineering or refactoring. From Table 10, 119 new classes were added, 11 of which (9.24%) were those classes removed from the system in the previous transition between versions 10 and 15. In addition, we found that between versions 15 and 20, 15 classes were removed from the system all of which again were inner classes. Anecdotal evidence would suggest that the existence of constructs such as Java inner classes influences the scrutinizing role of the developer by complicating the task of maintenance. Inner classes allow a nested class access to the attributes of the enclosing class and have been the subject of certain criticism since they add a level of complexity to the system (McGraw and Felten, 1998; Sintes, 2001).

4.4.6. Class changes (versions 20-22). In the transition from version 20 to 22 only 1 class was moved from DIT=6 to DIT 7 (this class was also moved from DIT=5 to DIT=6 between versions 15 and 20). 1 inner class was removed and overall 42 new classes were added of which 24 were added at DIT=1, 12 classes added at DIT=2 and 6 classes added at DIT=3. In SwingWT, the total number of moved classes was 133 of which only 6 (i.e., 4.51%) classes were inner classes. The total number of new classes in the system was 621 of which 141 (i.e., 22.71%) classes were inner classes, and the total number of removed classes was 52 of which 37 (i.e., 71.15%) classes were inner classes suggesting again that inner classes in SwingWT are far more amenable to deletion than 'regular' 'unenclosed' classes. Figure 7 shows the overall changes in SwingWT in the same format as Figure 4.

From Figure 7, we see that the majority of changes are increases in number of classes. Two peaks are visible in terms of number of moved classes between version 5 to 10 and 15 to 20. The trends in number of added, removed and moved classes contrast. Each seems to have a peak at different stages of evolution.

4.5 Tyrant

4.5.1. Summary detail. This system consisted of 122 classes in version 1 and grew to 273 classes by version 45. The maximum DIT for Tyrant was 5. Figure 8 shows the frequencies of number of classes at each DIT level in every fifth and final version of Tyrant. An interesting feature is that in version 4, the system underwent a major change. The maximum DIT dropped from 5 in version 4 to 3 in version 5. The number of classes at DIT=1 increased from 45 in version 4 to 96 in version 5. The number of classes at DIT=2 dropped from 42 to 13. Finally, the number of classes at DIT=3 increased from 22 in version 4 to 63 in version 5. Since we found that the system went through significant changes between versions 4 and 5, we therefore included that transition in our analysis.

4.5.2. Class changes (versions 1-4). Between versions 1 and 4, the total number of classes in the system increased from 122 to 143. In the transition from version 1 to 4, only five classes were relocated in the hierarchy. A single class was added at DIT=1 and 2 existing classes from DIT=1 were placed as its subclasses. In addition, 2 new classes were added at DIT=3 and 2 classes from DIT=3 were moved as subclasses of one of them. One class from DIT=3 was moved to become a subclass of the other. We found that no classes were removed from the system.

4.5.3. Class changes (versions 4-5). Table 11 shows the number of moved classes at each DIT level in Tyrant, as well as the number of added and removed classes between versions 4 and 5 in the same format as Table 10. From Table 11, 48 classes were moved within the inheritance hierarchy. The maximum DIT dropped from 5 to 3. The total number of classes at DIT=5 in version 4 was 5, all of which were then removed from the system. The total number of classes at DIT=4 was 29, 11 of which were moved up to DIT=1. The majority of classes (34 classes in total) were moved to DIT=1. We also found that 56 classes were repositioned across the hierarchy in the entire 45 versions of Tyrant, 48 (85.71%) of which were moved in the transition from version 4 to 5. The total number of new classes was 226, 110 (48.67%) of which were added between versions 1, 4 and 5. The total number of removed classes was 75, 60 (80%) of which were removed in that same transition (version 4 to 5). A major re-engineering initiative seems to have occurred between versions 4 and 5. Figure 9 shows the trend in change frequencies in Tyrant.

Following the re-engineering of the system and flattening of the hierarchy, the inheritance hierarchy in Tyrant stabilized in terms of movement of classes. Only 2 classes were moved from DIT=1 to DIT=2 between versions 25 and 30, and only 1 further class was moved from DIT=2 to DIT=1 between versions 40 and 45. The remaining changes were all either increases or decreases in the number of classes. This again supports the view of Daly et al., (1996) on the significance of inheritance at 3 levels.

5. Analysis of class characteristics

So far, we have investigated the trends in movement of classes within the existing inheritance hierarchy. The decision to move a class within the hierarchy may be influenced by the characteristics of specific classes. The reasons why classes may be moved can be cast in a number of ways. For example, large classes may be difficult to move and may require many other classes to be changed as a result. However, they may well offer other classes that need it the benefit of that functionality if moved and this could be a compelling argument for moving such classes. In contrast, moving small classes may require a significantly less amount of effort, but, on the other hand, they may contain and hence offer less functionality to other classes. In the same way, classes with high coupling may be harder to move than those loosely coupled, since highly coupled classes, by definition, have many dependencies. However, moving classes that highly coupled may re-locate classes to where they are needed and used most. We therefore conjecture that (i) larger classes are more likely to be moved across the hierarchy than smaller classes and (ii) classes with high coupling are more likely to be moved than loosely coupled classes (with few dependent classes).

We analysed characteristics of the moved and static classes in all four systems to determine whether this was actually the case in the four studied systems. We analyzed two features of the classes in the four systems (i) class size, given by number of methods (NOM) metric of Lorenz and Kidd (1994) and (ii) coupling given by the Message Passing Coupling (MPC) metric of Li and Henry (1993). We formed and tested the following hypotheses in order to investigate whether larger classes and tightly coupled classes were more frequently moved within the hierarchy.

The null hypothesis H01 states: Class movement and relocation is not influenced by class size.

The alternative hypothesis HA1 states: Larger classes, given by their NOM, are more likely to be moved within the hierarchy than smaller classes, as a system evolves.

The null hypothesis H02 states: Class movement and relocation is not influenced by class coupling.

The alternative hypothesis HA2 states: Tightly coupled classes, given by their MPC, are more likely to be moved within the hierarchy than loosely coupled classes as a system evolves.

Table 12 shows the maximum (Max.), mean, median and standard deviation (STDEV) of NOM and MPC for moved (prior to their movement) and un-moved classes in the four studied systems. From HSQLDB, all values (with the exception of the median NOM) of NOM and MPC for moved classes are relatively smaller than their corresponding values for un-moved classes, suggesting that classes with relatively low NOM and MPC values did tend to be favoured when moving classes across the inheritance hierarchy. The mean and median values of NOM for moved classes in JasperReports show a different trend to that of HSQLDB. If we consider the mean values in JasperReports, we see that classes with higher NOM values were moved within the inheritance hierarchy. We see that all MPC values for moved classes in JasperReports are smaller than their corresponding values for un-moved classes. This again implies that in JasperReports, classes with fewer coupling features were moved in the hierarchy. In SwingWT, we see that all values of NOM and MPC, with the exception of their Max. values, for moved classes are larger than their corresponding values for un-moved classes, suggesting that larger classes, given by NOM, and highly coupled classes, given by MPC, were more frequently moved within the system inheritance hierarchy. Furthermore, all values of NOM and MPC for moved classes in Tyrant, with the exception of Max and STDEV MPC, seem to be higher than their corresponding values for un-moved classes. This again implies that in Tyrant, larger classes and relatively highly coupled were moved across the hierarchy.

To formally test the hypotheses (H01 and H02), we carried out two one-tailed non-parametric Mann-Whitney U-tests on moved and un-moved classes and their NOM and MPC. Table 13 shows the results of the Mann-Whitney U-test carried out on moved and un-moved classes and NOM in the four systems.

From Table 13, there is a significant difference between the two samples (moved and un-moved classes) in the four systems. The mean rank value for moved classes is higher than that of un-moved classes, suggesting that larger classes, given by NOM, were more frequently moved within the hierarchy than smaller classes. The p-value for the test is < 0.01 i.e., the test is statistically significant at one percent level. Table 14 shows the results of the Mann-Whitney U-test carried out on moved and un-moved classes and MPC.

From Table 14, we again see a significant difference between the two samples. The mean rank value for moved classes is higher than that of un-moved classes in the four systems, suggesting that classes with higher coupling, given by MPC, were more frequently moved within the hierarchy than classes with lower coupling. The p-value for the test is < 0.01 i.e., the test is statistically significant at one percent level.

Based on the evidence in Tables 13, we see that in the four systems larger classes were more frequently moved within the hierarchy than smaller classes - We are therefore in the position to reject H01in the favour of HA1; larger classes are more likely to be moved within the hierarchy than smaller classes as a system evolves. In terms of coupling (Table 14), classes with high MPC were moved more frequently than classes with low MPC - We are therefore in the position to reject H02 in the favour of HA2; tightly coupled classes are more likely to be moved within the hierarchy than loosely coupled classes as a system evolves.

One possible explanation for movement of large and highly coupled classes may be the lack of cohesion in those classes. Smaller classes and loosely coupled classes may be more cohesive than larger classes and tightly coupled classes respectively - they therefore often remain un-moved because they have not deteriorated sufficiently to necessitate being moved. To investigate this feature, we conducted two widely used non-parametric cross-correlations (Kendall's and Spearman's) of size (given by NOM), coupling (given by MPC) and cohesion (given by LCOM). Table 15 shows the correlation values of NOM versus LCOM and MPC versus LCOM in the four systems.

From Table 15, all the values are double asterisked indicating that the correlations are significant at the one percent level. The correlation values suggest that there is a strong and significant relationship between NOM and LCOM, and MPC and LCOM in the four systems. This latter result implies that class size and coupling are negatively related to cohesion. This was an interesting result to emerge from our analysis and we believe that the key driver for frequent movement of larger and tightly coupled classes within the hierarchy is the lack of cohesion in those classes.

Class movement and re-location provide valuable information to software managers and developers on the dynamics of a system. In the study presented, we found that larger classes and highly coupled classes were more frequently moved within an inheritance hierarchy. While we have no information, as such, on exactly why this might be, we hypothesise that, often classes are moved from one location to another because it 'improves' the system structure in some way. For example, if a class is highly coupled in ways other than through the use of inheritance, the potential for faults may be reduced and system comprehension improved if that class is moved to a location where, as a direct result, coupling is correspondingly reduced. On the assumption that movement generally improves a system, but, nevertheless, we would still like to minimise the amount of 'moving' we do, identification and prioritisation of classes to be moved should be a key project goal and a measure of its quality.

Analysis of the source data used in the study also revealed that large classes and highly coupled classes were generally less cohesive, further pointing to the need to address the problems associated with these type of classes. Developers and project managers should attempt to minimise class size and coupling from the outset of a project to avoid the effort associated with later class re-location which becomes necessary as problems in the system structure emerge. As a result, cohesion as given by the LCOM metric should reflect more cohesive classes.

Movement of classes with higher NOM and those with higher MPC can also be justified on the basis that developers may reduce structural complexity by moving large classes and classes with many coupling dependencies. Table 16 shows the maximum (Max.), Mean, Median, and standard deviation (STDEV) values for classes after they had been moved. From Table 16, all values of NOM and MPC for HSQLDB, SwingWT and Tyrant (with the exception of median NOM) are considerably higher than the corresponding values presented in Table 12 (where the same data for moved classes, prior to their movement, is presented). This suggests a significant growth (in terms of NOM and MPC) in these systems. For JasperReports however, the opposite occurred and all values of NOM and MPC for moved classes are smaller than the corresponding values presented in Table 12. This was surprising considering the significant growth of the system (818 classes in version 1 and 1098 classes by version 12).

Given that larger classes and highly coupled classes were more frequently moved within the inheritance hierarchy, we speculate that class movement may improve class cohesion. That is, class movement may have positive implications on class cohesion. To investigate this phenomenon we formed and tested the following hypotheses.

The null hypothesis H03 states: Class cohesion is not influenced by class movement and relocation.

The alternative hypothesis HA3 states: Class movement and relocation improves class cohesion.

To formally test the hypothesis H03, we categorised moved classes into two groups a) before their movement and b) after their movement. We took class movement as the main unit of our analysis and conducted a Wilcoxon signed-rank test (a non-parametric test on two related samples) to identify the impact of that movement on class cohesion. Table 17 shows the results of the Wilcoxon signed-rank test carried out on moved classes before/after their movement and their cohesion.

From Table 17, we see that the 'After' variable appears first suggesting that it was first entered into the equation. The negative ranks therefore shows that 46 ranks of 'Before' were greater than 'After'. The positive ranks shows that 73 ranks of 'Before' were smaller than 'After'. Finally, the Tie ranks shows that 101 ranks of 'Before' and 'After' were equal. From table 17, we also see that the z-score and p-value are -0.873 and 0.383, respectively, implying that the test is not statistically significant. We therefore conclude that there is not significant difference between the cohesion of the classes before and after their movement and we thus accept H03: Class cohesion is not influenced by class movement and re-location.

6. Discussion

Our findings may be of interest to software managers in predicting how systems change over time, which types of classes are most frequently moved within an inheritance hierarchy, strategies for minimising those movements and as importantly, targeting re-engineering and refactoring effort on regularly changing parts of a system. If effort is consistently being demanded in specific parts of a system's hierarchy (as observed in our study), then this highlights the need for project managers and developers to investigate exactly why this is happening.

Also, the role of coupling cannot be under-stated. Identification of coupling patterns in classes that are consistently being moved may highlight which types of coupling facilitate movement - some forms of coupling may be less troublesome than others. Take the example of coupling in the form of parameter types of the methods of a class, called let's say *X*. This form of coupling can usually be moved with the class *X* without penalty since the method dependencies move with the class. At the other end of the scale, method invocation in other classes through an inheritance relationship may well leave dangling references if *X i*s moved without consideration of such relationships. In other words, the results of the study inform an understanding of which 'types' of coupling predominate in moved classes. Finally, if large classes (in terms of number of methods) do not seem to preclude the re-location of classes, then this means that size *per se* might not be a prime consideration for re-engineering effort. Imposing rules of thumb on the minimum and maximum number of allowable methods in a class (as some industry standards suggest) may be a short-sighted policy. Size may not be as important as other class characteristics.

The study also informs a number of ongoing research issues and presents a range of research opportunities. The results reinforce the notion that inheritance is not used to the depth that it was envisaged by early proponents of OO. The fact that some parts of the inheritance hierarchy collapsed during evolution suggests that inheritance depth may actually find its own level if the hierarchy is too deep or developers see advantages to moving classes up the hierarchy; it may be case that at design stage, relatively deep levels of inheritance are appropriate but as a system evolves, that appropriateness weakens. The research also highlights the existence of certain clusters of classes which are so inter-related that they can only be moved *en masse*. This presents opportunities for re-engineering and amalgamation of classes. Usually, we think about decomposition of classes as a prime re-engineering task; the study shows that the opposite can sometimes be as useful. In contrast, the research paves the way for an investigation of refactoring opportunities based on coupling and cohesion and their interplay. For example, we have found that cohesion in re-located classes does not change significantly. This might be because coupling is not a key part in the calculation of the LCOM metric. Research on the effect on these software features of refactoring, application of software metrics and sensitivity analysis of these measures to class movement is very limited.

7. Conclusions and future work

In this paper, we presented an empirical study of evolution of classes in four Java systems. Inheritance data was collected using the JHawk tool and changes observed in terms of newly added classes, deleted classes and moved classes within the inheritance hierarchies in multiple versions of the four systems. We suggest if developers are restricted to changes in just one part of the system, then that might reflect a poor design and/or poorly applied previous maintenance in that specific part of the system. In theory, changes should be evenly spread across all parts of the system, but in practice there seem to be 'hot spots' in a system, i.e., areas of code that require constant developer attention. Identifying where these areas tend to occur and, more importantly, why they occur, could help future effort to be directed and estimated.

That said, we are mindful of the fact that an 80/20 rule may apply to system change i.e., that focus of 80% of maintenance activity might be applicable to just 20% of system functionality or, in this case, 20% of the inheritance hierarchy and that changes are unrelated to bad design; they are to classes and parts of the hierarchy which, by their nature, need significant and consistent attention. Wheeldon and Counsell (2003) have shown that class relationships follow an 80/20 rule; equally, underpinning our work is the assumption that Lehman's Laws apply, when in fact there has been some empirical evidence to suggest that those laws do not follow and that growth of a system is unpredictable in this sense (Godfrey and Tu, 2001; Herraiz et al., 2007).

Further results from the study presented suggested that developers tended to focus most of their activity (from the change data) at, and above, level 3 rather than beyond level 3. We also found very little activity beyond level 3 of inheritance hierarchies in the studied systems (with the exception of SwingWT where 7 levels of inheritance were used). In addition, we found evidence of hierarchies being 'squashed' to bring classes up to shallower levels rather than them remaining at deep levels. Interestingly, we also found evidence to suggest that larger classes and tightly coupled classes were more frequently moved within the hierarchy than smaller classes and loosely coupled classes. We investigated the reasons why this may have occurred and found smaller classes and loosely coupled classes to be more cohesive than larger classes and tightly coupled classes; the movement of larger classes and tightly coupled classes may be due to the lack of cohesion in those classes. We believe these results are of some relevance, since software systems spend most of their `life' in maintenance mode. Understanding where this change tends to take place helps predict future maintenance activity and target scarce refactoring resources to areas where most benefit will be observed.

Several questions arise which might affect the validity of the study. First, the generalisability of the results could be questioned. We argue that the set of systems analyzed were from various application domains (ranging from a database system to a GUI framework and game engine) with different sizes. Second, the study may be criticised for using OSS and not proprietary software. We argue that both OO and OSS has recently been the subject of a great deal of empirical research (Advani et al., 2006; Capiluppi et al., 2004; Capiluppi and Ramil, 2004) and yet a wide range of research issues and research questions remain unanswered. From a practical viewpoint, the results of the study are of relevance to developers as to how inheritance hierarchies evolve in terms of incremental changes as well as movement of classes. The study contributes to a body of knowledge into evolution of OO systems at a finer-grain. To our knowledge, no previous studies have analysed the evolutionary behaviour of systems at lower levels of granularity.

In terms of future work, we would like to firstly, analyze systems from a refactoring perspective. In other words, to explore the possibility that consistent change in specific areas of a system can be mitigated by the use of refactoring. In addition, we would like to analyze the evolution of the systems at the attribute level, which may reveal insights into systems that analysis at the method level does not afford. We would advocate more studies of the type described in this paper and, to that end, all data used in this study can be made available to other researchers and practitioners.

8. Acknowledgements

We thank the anonymous referees for their insightful comments and suggestions for improving the paper and from which it has benefited significantly.

References

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.

Arisholm, E. and Briand, L.C., Predicting fault-prone components in a Java legacy system, ACM/IEEE Intl. Symp. Empirical Soft. Eng., Rio de Janeiro, pp.8-17, 2006.

Arisholm, E. and Briand, L.C. and Fuglerud, M. J., Data Mining Techniques for Building Fault-proneness Models in Telecom Java Software. Simula Research Laboratory (01-2007) 2007.

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

Basili, V.R., Briand, L.C. and Melo, W.L., A validation of object-oriented design metrics as quality indicators, IEEE Trans. on Software Eng., vol. 22, no. 10, pp. 751-761, 1996.

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.

Capiluppi, A., Morisio, M., and Ramil, J., Structural Evolution of an Open Source System: A Case Study, Proc. of the 12th Intl. Workshop on Prog. Comprehension, Bari, Italy, pp. 172-182, 2004.

Capiluppi, A. and Ramil, J., Studying the evolution of open source systems at different levels of granularity: Two case studies, Proceedings of 7th International Workshop on Principles of Software Evolution. Kyoto, Japan, pp. 113-118, 2004

Cartwright, M., and Shepperd, M., An Empirical Investigation of an object-oriented (OO) system. IEEE Trans. on Software Eng., 26(8), pp. 786-796. 2000.

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

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

Counsell, S., Loizou, G., and Najjar, R., Quality of manual data collection in Java software: an empirical investigation. Empirical Software Engineering 12(3), pp. 275-293. 2007.

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.

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

Girba, T., Lanza, M. and Ducasse, S., Characterizing the Evolution of Class Hierarchies, Ninth European Conf. on Software Maintenance and Reengineering, Manchester UK. pp. 2-11 2005.

Godfrey, M and Tu, Q. Growth, evolution, and structural change in open source software, Proceedings of the 4th International Workshop on Principles of Software Evolution, Vienna, Austria, pages 103-106, 2001.

Harrison, R., Counsell, S., Nithi, R. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *Journal of Systems and Software*, 52(2-3): pp.173-179, 2000.

Herraiz, I. Gonzalez-Barahona, J., Robles, G., and German, D.M. On the prediction of the evolution of libre software projects. IEEE International Conference on Software Maintenance, Paris, France, pages 405-414, 2007.

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

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

Lehman, M.M., Programs, Cities, Students, Limits to Growth? Inaugural Lecture, May 1974, *published in Imperial College of Science and Technology. Inaugural Lecture Series*, vol. 9, pp. 211-229. 1970-1074, 1974. *Also in Programming Methodology, D. Gries Ed* New York: Springer-Verlag, pp.42-69, 1979.

Lehman, M., Ramil, J., Wernick, P., Perry, D., and Turski, W. M., Metrics and Laws of Software Evolution - The Nineties View, IEEE Intl. Symposium on Software Metrics (METRICS 97), Albequerque, USA, pages 20-32, 1997.

Li, W. and Henry, S., Object-oriented metrics that predict maintainability, The Journal of Systems & Software, 23 (2), pp. 111-122. 1993.

Lorenz, M. and Kidd, J. (1994) *Object-Oriented Software Metrics*. New Jersey: Prentice Hall

McGraw, G. and Felten, E., Twelve rules for developing more secure Java. Java World, 12/01/1998. available at: http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html?page=1.

Najjar, R., Counsell, S., Loizou, G. and Mannock, K., The Role of Constructors in the Context of Refactoring Object-Oriented Systems, Proc. of: 7th European Conference on Software Maintenance and Reengineering. Benevento, Italy. pp. 111-120, 2003.

Nasseri, E., Counsell, S., An Empirical study of Java System Evolution at the Method Level, To appear in the, Proc. of the 7th IEEE International Conference on Software Engineering Research, Management and application, Haikou, Hainan Island, China, 2009.

Nasseri, E., Counsell, S., and Shepperd, M., An Empirical Study of Evolution of inheritance in Java OSS, Proc. of: 19th Australian Software Eng. Conference, Perth, Australia, pp. 269-278, 2008.

Prechelt, L., Unger, B., Philippsen, M., and Tichy, W., A controlled experiment on inheritance depth as a cost factor for code maintenance, Journal of Systems and Software, 65(2):115-126, 2003.
Sintes, T. So what are inner classes good for anyway? Java World, 27/07/01. available at: http://www.javaworld.com/javaworld/javaqa/2000-03/02-qa-innerclass.html?page=1.
Wheeldon, R. and S. Counsell, Powel Law Distributions in Class Relationships, IEEE Source Code Analysis and Manpulation, (SCAM), Amsterdam, pages 45-54, 2003.
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.

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

System	No. of versions studied (n)	Max (Ch)	Norm.	Min (Ch)	Var (Ch)	Median (Ch)	Mean (Ch)
HSQLDB	3	176	271%	0	15336	23.5	58.6
Jasper Reports	4	183	22%	-77	11696	13.5	23.3
SwingWT	6	160	320%	0	39327	20.5	27.19
Tyrant	11	103	84%	-85	1657	0	3.58

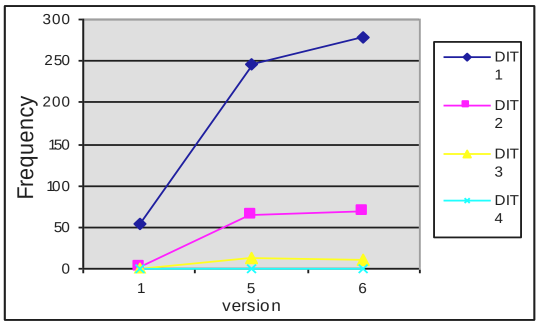

Figure 1. Number of Classes (HSQLDB)

Table 2. Class evolution between version 1 to 5 in HSQLDB

	DIT1	DIT2	DIT3	DIT4	RemC.	NewC.
DIT1	•	3	0	0	8	203
DIT2	0	•	0	0	0	59
DIT3	0	0	•	0	0	12
DIT4	0	0	0	•	0	1

Table 3. Class evolution between versions 5 to 6 in HSQLDB

	DIT1	DIT2	DIT3	DIT4	RemC.	NewC.
DIT1	⬩	1	0	0	8	39
DIT2	3	⬩	0	0	0	5
DIT3	0	1	⬩	0	0	0
DIT4	0	0	0	⬩	1	0

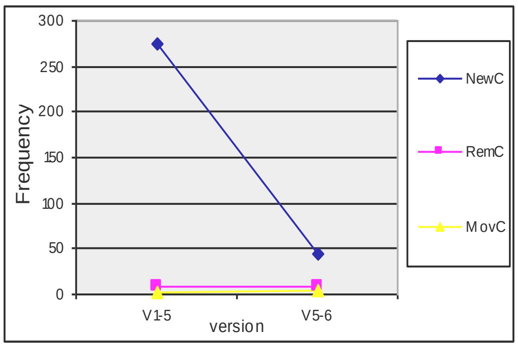

Figure 2. Changes in HSQLDB

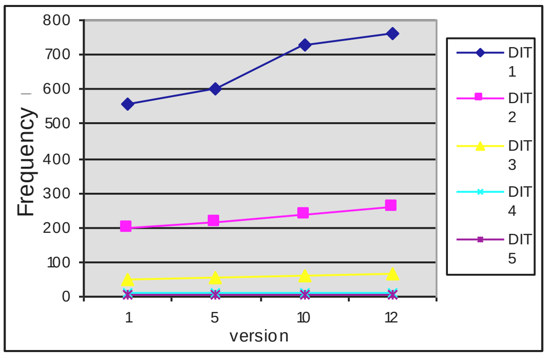

Figure 3. Number of Classes (JasperReports)

Table 4. Class evolution between version 1 to 5 in JasperReports

	DIT1	DIT2	DIT3	DIT4	DIT5	RemC.	NewC.
DIT1	•	4	1	0	0	8	59
DIT2	0	•	0	0	0	2	17
DIT3	0	0	•	0	0	0	3
DIT4	0	0	0	•	0	0	0
DIT5	0	0	0	0	•	0	0

Table 5. Class evolution between version 5 to 10 in JasperReports

	DIT1	DIT2	DIT3	DIT4	DIT5	RemC.	NewC.
DIT1	•	2	0	0	0	5	134
DIT2	0	•	0	0	0	0	23
DIT3	0	0	•	0	0	0	7
DIT4	0	0	0	•	0	0	0
DIT5	0	0	0	0	•	0	0

Table 6. Class evolution between version 10 to 12 in JasperReports

	DIT1	DIT2	DIT3	DIT4	DIT5	RemC.	NewC.
DIT1	•	12	0	0	0	1	45
DIT2	0	•	1	0	0	0	6
DIT3	0	1	•	0	0	0	2
DIT4	0	0	0	•	0	0	0
DIT5	0	0	0	0	•	0	0

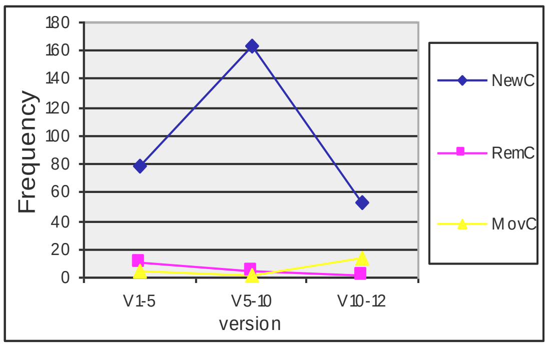

Figure 4. Changes in JasperReports

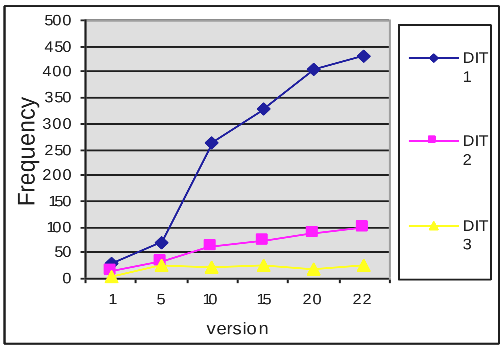

Figure 5. Number of Classes (SwingWT)

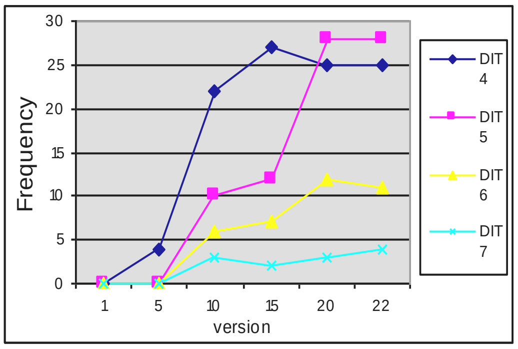

Figure 6. Number of Classes (SwingWT)

Table 7. Class evolution between version 1 to 5 in SwingWT

	DIT1	DIT2	DIT3	DIT4	DIT5	DIT6	DIT7	RemC.	NewC.
DIT1	•	0	0	0	0	0	0	1	42
DIT2	0	•	0	0	0	0	0	1	17
DIT3	0	1	•	0	0	0	0	0	20
DIT4	0	0	0	•	0	0	0	0	4
DIT5	0	0	0	0	•	0	0	0	0
DIT6	0	0	0	0	0	•	0	0	0
DIT7	0	0	0	0	0	0	•	0	0

Table 8. Class evolution between version 5 to 10 in SwingWT

	DIT1	DIT2	DIT3	DIT4	DIT5	DIT6	DIT7	DelC.	NewC.
DIT1	◆	0	0	1	0	0	0	2	196
DIT2	0	◆	10	3	3	0	0	0	45
DIT3	0	1	◆	11	4	6	0	0	11
DIT4	0	0	0	◆	1	0	2	0	5
DIT5	0	0	0	0	◆	0	0	0	2
DIT6	0	0	0	0	0	◆	0	0	0
DIT7	0	0	0	0	0	0	◆	0	1

Table 9. Class evolution between version 10 to 15 in SwingWT

	DIT1	DIT2	DIT3	DIT4	DIT5	DIT6	DIT7	RemC.	NewC.
DIT1	◆	2	3	0	0	0	0	23	90
DIT2	0	◆	0	2	0	0	0	6	15
DIT3	1	0	◆	3	0	0	0	0	3
DIT4	0	1	0	◆	2	0	0	3	6
DIT5	3	0	0	0	◆	0	0	0	1
DIT6	0	0	0	0	0	◆	0	0	1
DIT7	0	0	0	0	2	0	◆	0	1

Table 10. Class evolution between versions 15 to 20 in SwingWT

	DIT1	DIT2	DIT3	DIT4	DIT5	DIT6	DIT7	RemC.	NewC.
DIT1	◆	1	0	0	8	0	0	12	96
DIT2	0	◆	1	0	0	2	0	2	15
DIT3	0	1	◆	12	0	1	1	0	1
DIT4	2	1	4	◆	14	1	0	1	3
DIT5	0	0	1	6	◆	5	0	0	3
DIT6	0	1	1	0	3	◆	2	0	1
DIT7	0	0	0	0	0	2	◆	0	0

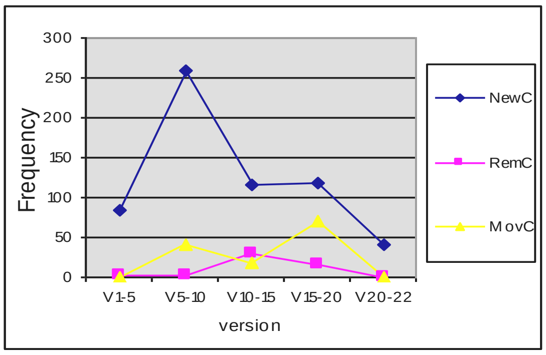

Figure 7. Changes in SwingWT

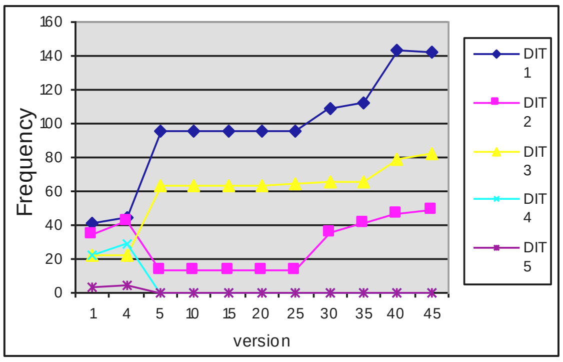

Figure 8. Number of Classes (Tyrant)

Table 11. Class evolution between version 4 to 5 in Tyrant

	DIT1	DIT2	DIT3	RemC.	NewC.
DIT1	◆	3	1	11	32
DIT2	12	◆	8	18	4
DIT3	11	1	◆	9	53
DIT4	11	1	0	17	0

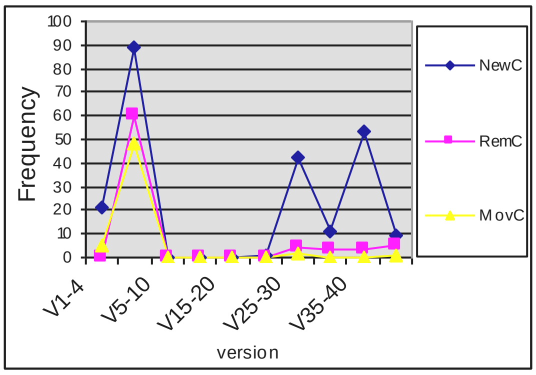

Figure 9. Changes in Tyrant

Table 12. Class characteristics in the four systems

		Moved Classes				Un-moved Classes			
		Max.	Mean	Median	STDEV	Max.	Mean	Median	STDEV
HSQLDB	NOM	14	7.25	7	4.95	171	12.57	7	17.90
	MPC	73	23.37	9.5	30.08	593	42.32	16	73.04
Jasper Reports	NOM	49	12.67	15	11.05	166	9.84	4	19.41
	MPC	74	7.76	0	19.08	1238	23.15	5	69.29
SwingWT	NOM	102	14.33	9	15.93	170	9.09	4	13.86
	MPC	224	21.11	10	33.68	279	11.16	0	26.14
Tyrant	NOM	88	12.96	9	14.65	63	6.27	2	9.84
	MPC	151	38.62	25	42.63	268	29.86	13	45.08

Table 13. Mann-Whitney U-test for moved/un-moved classes and NOM (all systems)

Classes	N	Mean-Rank	Sum of Rank	M-Whitney-U	p-Value	Z-Score
Moved	220	1472.28	323902.5	174727.5	0.000	-6.474
Un-Moved	2156	1159.54	2499973.5	X	X	X

Table 14. Mann-Whitney U-test for moved/un-moved classes and MPC (all systems)

Variables	N	Mean-Rank	Sum of Rank	M-Whitney-U	p-Value	Z-Score
Moved	220	1305.29	287163.5	211466.5	0.0035	-2.679
Un-Moved	2156	1176.58	2536712.5	X	X	X

Table 15. Correlations of NOM/MPC and LCOM

	Kendall's	Spearman's
NOM vs. LCOM	0.246**	0.310**
MPC vs. LCOM	0.255**	0.333**

Table 16. Descriptive statistics for moved classes after re-location

		Moved classes after re-location			
		Max.	Mean	Median	STDEV
HSQLDB	NOM	134	26.63	9	44.64
	MPC	498	93	13	171.30
JasperReports	NOM	48	6.38	4	10.10
	MPC	23	3.62	0	7.62
SwingWT	NOM	123	20.54	17	19.39
	MPC	278	32.67	19	44.77
Tyrant	MPC	127	14.30	8	21.70
	NOM	365	65.36	39.5	71.02

Table 17. Wilcoxon signed-rank test for moved classes before and after their re-location

		N	Mean Rank	Sum of Ranks	Z-Score	P-Value
After - Before	Negative Ranks	46[a]	70.46	3241	-0.873[a]	0.383
•	Positive Ranks	73[b]	53.41	3899	•	•
•	Ties	101[c]	•	•	•	•
•	Total	220	•	•	•	•

a After < Before
b After > Before
c After = Before