Journal of Object Technology
Published by ETH Zurich, Chair of Software Engineering © Jot, 2010
Online at http://www.jot.fm

Size, Inheritance, Change and Fault-proneness in C# software

Matt Gatrell[a] **Steve Counsell**[a]

a. Dept. Information Systems and Computing, Brunel University

Abstract This paper documents a study of change in commercial, proprietary C# software and attempts to determine whether a relationship exists between class changes and faults and the design context of a class, namely its size and inheritance relationships. Results showed a strong positive correlation between the size of a class and change-proneness but not for all the class features studied. Classes within a specific range of a) inheritance depth and b) number of children were found to be relatively more prone to change. For the fault data and for the same class features, similar results were found. The most striking result to emerge however was the existence of an inheritance depth 'interval' between which change (and fault-proneness) were at their highest. Below and above that interval, both features were less prominent. The results thus add weight to the claims of other previous studies which suggest that there is an optimal level of inheritance, beyond which maintenance may become problematic from both a change and fault perspective.

Keywords C#, class changes, faults, inheritance

1 Introduction

Identifying changes made to a system over time can help identify problem areas and also inform remedial action by a developer and/or project manager. Such data can also be instrumental in helping to predict future change, the prioritization of work and the allocation of limited resources. The same potential benefits are true of fault data in its role of highlighting problematic areas of code and possible directed re-engineering or refactoring effort [Arisholm06, Fowler99]. The extent to

M. Gatrell and S. Counsell. Size, Inheritance, Change and Fault-proneness in C# software. In Journal of Object Technology, vol. 9, no. 5, 2010, pages 29-54.

Available at http://www.jot.fm/contents/issue_2010_09/article2.html

which these benefits can be realized is an issue of significant interest for exploratory, empirical software engineering studies. A number of studies have also cast doubt on the extent to which deep levels of inheritance assist with the maintainability of a system [Prechelt03, Cartwright00, Harrison97, Basili96].

In this paper, we analyze change through the design context of a class. More specifically, we explore change through the size of a class and its inheritance characteristics. We also explore fault data for the same system and the same two design perspectives (of size and inheritance). The basis of the study is a commercial C# system, consisting of 266K lines of code, 7,439 classes and 79,964 methods. The system had been subject to 19,054 changes over a two year period - these changes were due to both enhancements and fault fixing. Inheritance properties of each class were also identified based on their inheritance depth and the number of subclasses (i.e., children) belonging to each. The size and inheritance characteristics were compared to the change history of the classes to determine any relationships. Fault data over a later period was also analyzed to support the analysis related to change-proneness.

Results from our study showed that first, size had a strong positive influence on the propensity for a class to be changed; the same result was found when we studied the fault data. Analysis also revealed that beyond a certain inheritance depth, the propensity for change rose before declining; analysis of faults also showed the same phenomenon. The study therefore raises important development questions such as: should a limit be placed on the size of a class if large classes exhibit these features? Just as salient is the question as to whether developers should avoid extending inheritance hierarchies beyond a certain depth and width if they both exhibit a high change and fault-proneness? Finally, what bearing does knowledge of faults in a system have on our ability to understand change-proneness?

The remainder of the paper is organized as follows. In the following section, we describe the motivation for the study and related work. In Section 3, we describe preliminaries such as the system studied and metrics extracted by the tool we used for the study. We then analyze the data exploring the change-proneness of the data (Section 4). In Section 5, we examine the fault data for the same system. We then discuss several issues raised by the study as well its threats to validity (Section 6) before providing a discussion of the issues raised (Section 7). Finally, we conclude and discuss further work in Section 8.

2 Motivation and related work

The motivation for this research arises from three sources. First, to our knowledge earlier studies that have shown a relationship between class size and inheritance

properties with change and fault data have not been replicated frequently, yet in the case of class size they represent studies that can be replicated relatively easily. The analysis of faults in our study extends analyses that have previously just looked at class size and earlier preliminary work by the authors on design patterns and their change-proneness [Gatrell09b, Bieman01, Bieman03]. Second, the controversy over the use of inheritance in OO has raged for over fifteen years – and yet we still know very little about whether, and to what extent, using a deep or wide inheritance hierarchy limits or impairs the maintenance process. In this study, we look at both change and fault-proneness in our analysis and therefore address this issue from two inter-related sides. Third, while there have been a number of studies exploring the shape of an inheritance hierarchy, there is no consensus on the extent to which an inheritance hierarchy should be structured on a width-basis and, as relevantly, the merits of using inheritance width over depth.

Two key previous studies are of particular relevance to our analysis. In Bieman et al. [Bieman01], the effect that size and inheritance characteristics had on change in 39 versions of a large C++ system was shown. The study found that large classes were the most change-prone. Results from a later study by Bieman et al. [Bieman03] using C++ and Java systems were largely inconclusive with respect to class size and change proneness. Only for two of the systems were large classes more change-prone. The same study also observed counter-intuitive characteristics of the inheritance hierarchy; classes at level zero (the root of a hierarchy) were changed more often than classes at level 1 and 2 of the inheritance hierarchy. In this paper, we explore the same research question with respect to size and class change addressed in both previous studies.

The role that the 'depth' of inheritance plays in the context of this paper is highly significant. Many studies have analyzed inheritance in OO systems and most have cast doubt on the use of deep inheritance hierarchies. The Depth of Inheritance Tree (DIT) metric, originally introduced by Chidamber and Kemerer (C&K) [Chidamber94] has been used in many empirical studies investigating inheritance structures. Many studies have reported a lack of use of inheritance to a deep level while others have reported a problem emerging below a certain level. Moreover, only limited numbers of studies have explored the relationship between DIT and faults. Basili et al. [Basili96] was one study that used the C&K metrics as predictors of fault-prone classes. Data from eight medium-sized C++ management systems were collected. Statistically significant results suggested that a class located deep in the inheritance hierarchy (given by its DIT) was more fault-prone than a class higher up in the hierarchy; the study suggested that extensive use of inheritance could have had the opposite effect to that of aiding the maintenance process. Prechelt et al. [Prechelt03] suggest that maintenance effort is positively associated with inheritance depth (i.e., the deeper the inheritance hierarchy, the more maintenance effort required – and this would suggest that this is where the potential for faults to be invested lay). Wood et al. [Wood99] advise

that inheritance should be used with care and only when needed. Bieman and Zhao [Bieman95] describe a study of nineteen C++ systems, comprising 2,744 classes in total. They found that only 37% of the systems had a median class inheritance depth greater than one. Cartwright and Shepperd [Cartwright00] describe the collection of a subset of the C&K metrics from a large telecommunications subsystem (133,000 lines of C++) and 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. The lack of adherence to 'expert' advice on the use of inheritance is further noted in the work of [Gorschek10] in a large-scale study of OO practitioners.

The experience of software engineers and researchers therefore seems to imply that deep levels of inheritance should be discouraged and they might actually be the source of maintenance problems rather than an aid to maintenance. The study presents additional empirical evidence that inheritance to a deep level (as well as to a large width) might be counter-productive in terms of change and fault-proneness. In the next section, we describe preliminaries to our study.

3 Preliminaries

3.1 The software system analyzed

The system used as a basis of the empirical study, 'WebCSC', was written by a large, international software company specializing in transaction content processing software. One of the authors was an architect in the company and had access to the version control system and hence to the change and fault data for the system. The need to document and check-in every change was a standard imposed rigidly in the company and so we have some confidence in the veracity of the change and fault data we used. The code itself related to a core technology product written in C# by a team of 8-10 developers and had been running for approximately 4 years. The period over which our change analysis is based represents the most recent two years of its development. The fault data, on the other hand, relates to the most recent year of the system, since the fault reporting process had not been automated until that point. The system itself included server side components, a web application, a number of client side components and tools. WebCSC comprised over 7,439 classes and approximately 266K lines of code (LOC). Each modification in the version control system, whether for a fault fix, enhancement (or otherwise) constituted a new version for the class and each version was counted as a single change. For the purpose of this study and to align the study with that of previous studies, we assume that each change made to code by a developer is equivalent,

i.e., the relative size of the change in terms of LOC required by the change was not considered. (Consideration of this aspect of the analysis is a significant, yet complementary study and one that we therefore have to leave for future work.)

3.2 Size measures

For the purpose of this study, size was measured by LOC, Number of instance methods in a class, Number of static methods in a class and Number of fields in a class. We also collected Total number of methods, Number of properties (defined as 'getters' or 'setters' of a field in C#) and Number of operations (defined as the sum of the number of class fields and methods). These metrics are in keeping with the earlier studies of Bieman et al. [Bieman01, Bieman03]. A bespoke tool written by the authors was used to extract this information from the latest version of the WebCSC system - the version control system contained all changes made to every class since its inception.

We note that LOC only considers executable code. Declarations were not counted, nor were interfaces, abstract methods or enumerations. Comments were also ignored and where a single logical LOC was spread over multiple lines for coding style, e.g., there were a large number of arguments to a method call, only a single LOC was counted.

3.3 Inheritance properties

The inheritance properties of classes were measured through the DIT and the Number of Children (NOC) belonging to class metrics. Both of these metrics, originally defined by C&K [Chidamber94], have been used extensively in empirical studies since [Basili96, Daly96, Harrison97]. DIT was collected by considering each class in the WebCSC system and determining the maximum length of the path from the class to its root class. The NOC was collected for each class by determining the number of immediate subclasses (note, we use the term 'subclass' and 'child' inter-changeably in this paper). The DIT and NOC metrics were extracted using a plug-in to the build server for the WebCSC system.

4 Change analysis

During the two year period, a total of 19,054 changes were made to the system. 4,434 of the 7,439 classes had no changes made to them at all over the same period.

A large number of classes had between 2 and 30 changes made, and only 29 classes had had 30 or more changes. The most frequently changed class had had 145 changes applied to it, nearly double the number of changes of the second most change-prone class, with 75 changes. Table 1 shows the frequency of changes per class and shows that 29 classes had over 30 changes, 56 had between 20 and 29 changes, and 280 had had between 10 and 19 changes applied to them.

Table 1- Number of changes/class

0 changes	1-9	10-19	0-29	>= 30
4434	2687	280	56	29

The high bias towards classes having less than ten changes can be seen from Table 1; in total, 4,434 classes had had no changes at all applied to them and 2,687 had had between 1 and 9 changes. The mean number of changes per class in the system was 2.55.

4.1 Hypotheses H1-H3

Three hypotheses were explored as part of our study of changes made to the WebCSC system. We note that the first hypothesis is identical in composition and wording to that originally posed by Bieman et al. in [Bieman01]; the second has been changed marginally from that also proposed by the same study to read more succinctly and clearly and the third hypotheses is one that we investigate independently.

Hypothesis H1: Are larger classes more change prone? A larger class has more functionality and there is therefore a greater likelihood that some functionality in the class will need to be corrected or enhanced.

Hypothesis H2: Classes located high up in an inheritance hierarchy will be more change-prone. Such a class has more dependents and there is therefore a greater likelihood that some functionality in the class will need to be enhanced because of changing requirements in those dependent classes.

In other words, the use of specialization in an inheritance hierarchy places a responsibility on classes high up in the hierarchy to provide appropriate functionality to subclasses as a part of requirements change also.

Hypothesis H3: Classes with a large number of children (subclasses) will be more change-prone than other classes. This hypothesis is based on the belief that a class with many children will be the subject of greater maintenance activity, since there are added dependencies on the parent class because of the changing requirements of a large number of children.

4.1.1 Class size and change (H1)

In this paper, each of a set of class size measures was correlated against number of changes. Figures 1 to 7 show the relationship between number of changes and each of those size measures. From inspection of these figures we see that all of: LOC, Number of instance methods, Number of static methods, Number of fields and Total number of operations are strongly correlated to change-proneness.

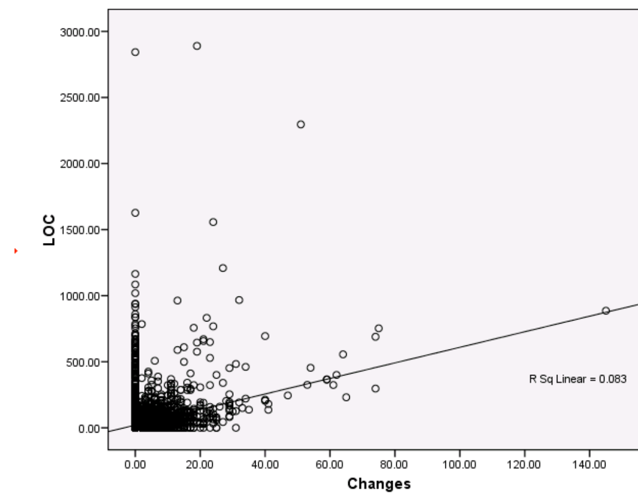

Figure 1 - LOC vs. number of changes

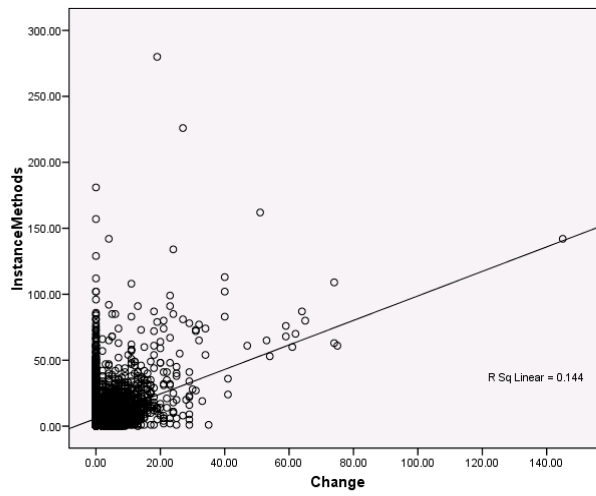

Figure 2 - Instance methods vs. changes

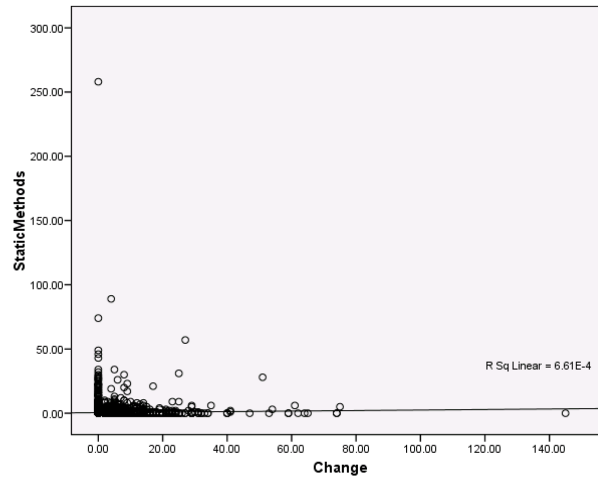

Figure 3 - Static methods vs. changes

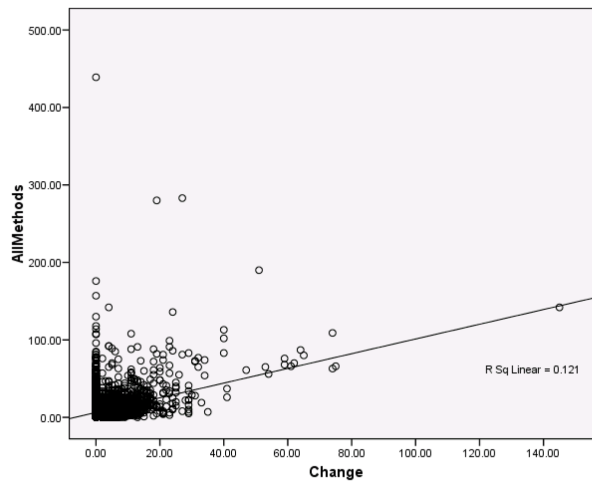

Figure 4 - Methods vs. changes

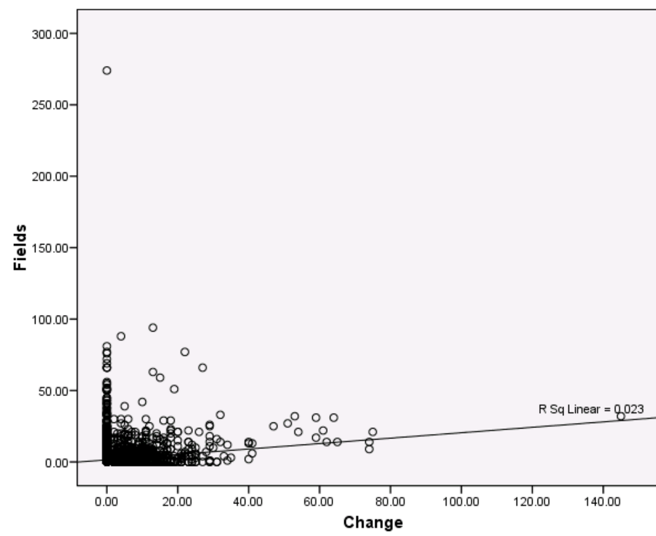

Figure 5 - Fields vs. changes

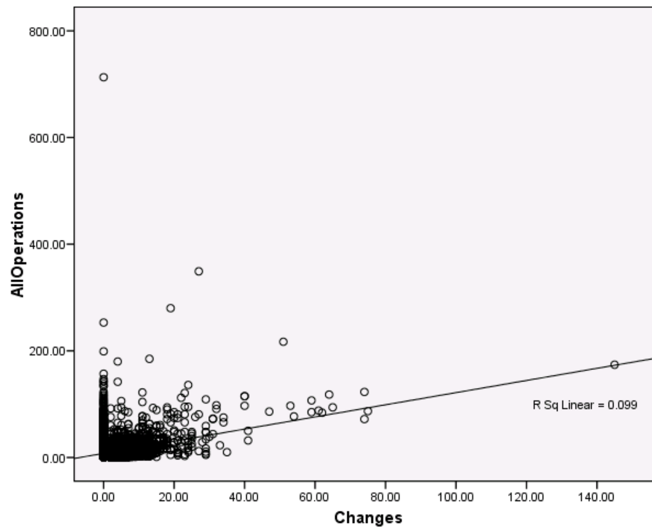

Figure 6 - Operations vs. changes

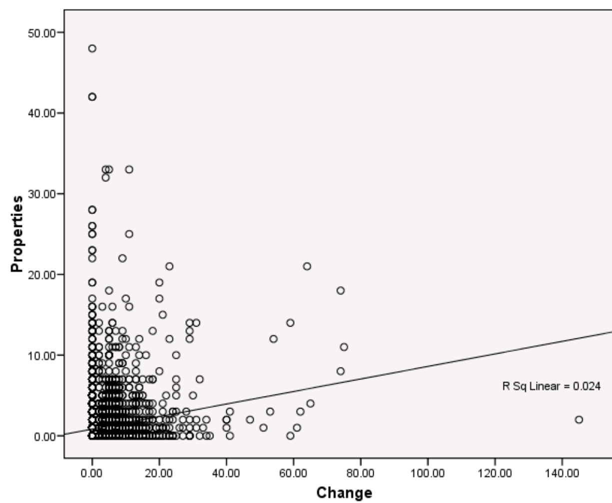

Figure 7 - Properties vs. changes

Table 2 shows the statistical correlation values for each class size metric versus change. A single '*' represents statistical significance at the 1% level and a '**' asterisk, significance at the 5% level. We computed both Pearson coefficients (a parametric test) and Spearman rank (a non-parametric test) correlation coefficients [Field05]. The values for Pearson's coefficients show that all size measurements had a strong influence on the propensity for change. Spearman's rank correlation values show that all size measurements (except number of static methods and

number of fields) also had a strong relationship with change propensity. The number of static methods has the weakest relationship (Pearson's value of 0.03), while the number of instance methods has the strongest relationship change (Pearson's value of 0.38).

One explanation for the lack of correlation found between static methods and change is that the latter do not actually use instance variables – so, in theory, they may be less likely to be modified as regularly as methods containing instance variables. In other words, we would expect classes where no instance variables are being manipulated to be modified less than classes that do (other things remaining equal). One suggestion for the relative lack of correlation for fields is that it is not often that fields (in either their name or declaration) are changed – it is the functionality that manipulates those fields which we would expect to comprise the bulk of changes made to a class. In other words, what the methods do with those instance variables is (on average) likely to be more change-prone than modification of the declared instance variables themselves.

The strongest correlation (in terms of Spearman's coefficient) was for LOC. Since the more methods, the greater the number of LOC, we would expect an explicit size measure to correlate with any feature of a class. In Bieman et al's study [Bieman03], the relationship between operations and changes was found to be stronger than that between fields and changes and from Table 2 the same relationship appears to be the case in our study. In contrast to the earlier study however, where operations were found to have the strongest relationship, our study indicates that a number of features (LOC, instance methods, total methods and properties) all correlate more strongly than that for operations.

Table 2 - Change correlation coefficients

Metric	Pearson's	Spearman's
LOC	0.29*	0.18*
Operations	0.32*	0.14*
Instance methods	0.38*	0.16*
Static methods	0.03**	0.00
Methods	0.35*	0.17*
Properties	0.16*	0.17*
Fields	0.15*	-0.02

In terms of the original hypothesis, we can clearly find strong support for H1 in light of the evidence presented. Large classes are more change-prone; however, in

keeping with the earlier result reported in [Bieman03] and in contrast with [Bieman01], we do not find overwhelming support for the hypothesis. We need to be mindful of the fact that there are size features of a class in the case of the system studied that might reduce the chances of those feature needing to be changed – the role of static methods and fields are cases in point here. We also posit that each system is likely to have its own idiosyncrasies that might cause it to exhibit slightly different correlation features than other systems based purely on the different nature of its application domain.

4.1.2 DIT analysis (H2)

The second hypothesis (H2) we explore is whether 'classes located high up in an inheritance hierarchy are more change-prone than those lower down?' To determine the extent to which inheritance characteristics influenced class change, we again used the class-based DIT and NOC metrics of C&K as a vehicle [Chidamber94].

Figure 8 shows the change profile for classes at different DIT levels. The system mean change value appears as a base line value of 2.55. The majority of classes had a DIT range of between 0 and 3. Classes in this category all had a similar rate of change to the mean of change for the entire system. However, classes with a DIT in excess of 3 had a higher rate of change than the system average. Many studies of inheritance have shown that systems typically have a very low median DIT value [Bieman95, Cartwright00, Nasseri08]. Several studies have also suggested that DIT 3 is the threshold level before inheritance becomes unwieldy and impractical to use [Daly96, Harrison97] and the evidence here seems to support that trend. Figure 8 shows that there is a clear peak in the propensity for class change at DIT 5 and 6. In fact, the number of changes rises rapidly after DIT 3, and then peaks at DIT 6, before decreasing again.

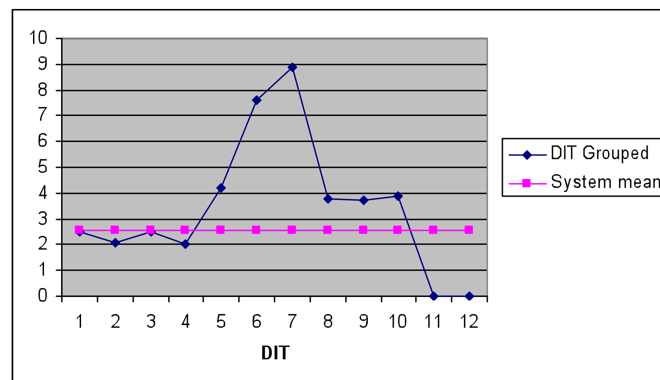

Figure 8 - Mean changes per DIT

Table 3 shows graphically the mean changes per class (Mean Change) grouped according to DIT level. The peak of 8.88 can be seen at DIT 6 after which the mean change falls significantly. The result for DIT indicates that there is an interval (the shaded region in Table 3) where relatively fewer changes are made to classes both below it and above it. One criticism of this analysis might be that since the majority of classes are likely to be at DIT 0-3, then that is inevitably where the changes will take place. However, since we are observing mean values, this does not explain the high values at DIT 4-6, or indeed the sudden fall at DIT 7. From the preceding analysis, we find little support for H2 from the data. It is not true that classes located high up in an inheritance hierarchy are more change-prone. It is actually in the middle tiers of the hierarchy where most changes on average are made.

Table 3 - Changes/class grouped by DIT

DIT	Classes	Mean change
0	1116	2.65
1	2445	2.14
2	1836	2.64
3	1362	1.98
4	296	4.39
5	92	7.38
6	41	8.88
7	214	3.29
8	52	3.19
9	31	3.52
10	1	0
11	1	0

4.1.3 NOC analysis (H3)

While the DIT metric provides a useful profile of one aspect of the inheritance hierarchy, it does not provide a view in any sense of the width of the hierarchy. The NOC provides this feature and allows us to establish whether a relationship exists between change and inheritance width. The NOC values collected from the

WebCSC system showed that the majority of classes had zero immediate subclasses. Those classes had a similar rate of change to the mean of the entire system (i.e., 2.55 changes). Classes with an NOC between 1 and 20 however, had a higher rate of change than the mean of the system. Classes with an NOC in excess of 20 were found to have a proneness to change similar to the mean of the system.

In keeping with the findings for DIT therefore, a clear pattern emerges of classes below and above a certain interval of NOC being more change-prone than those either side of that interval. Table 4 shows this effect clearly. Statistical analysis showed a Pearson correlation coefficient of 0.03 (significant at the 5% level) and a Spearman rank correlation coefficient of 0.16, significant at the 1% level (for number of changes vs. NOC in each case).

Table 4 - Changes/class grouped by NOC

NOC	Classes	Max. Ch.	Mean Ch.
0	5951	145	2.30
1-10	1412	75	3.36
11-20	76	25	4.14
>20	238	74	2.79

In terms of the original hypotheses H3, there is an interval phenomenon being exhibited, outside of which change is fairly consistent. Within that interval however, change is significantly higher. We cannot therefore find support for H3 and it is not necessarily true that classes with a large number of children will be more change-prone than other classes. We did find evidence of a range where this was the case, however.

4.1.4 An explanation

One theory to explain this interval feature of inheritance found for DIT is that there is, at some point, a ‘cognitive tipping’ effect in evidence. In other words, up until a certain level of complexity, change is relatively easy (this is the case where few descendents above need to be considered and DIT is low). Beyond that level, it becomes difficult when both many ascendants below and many descendents above may need to be considered for any change. At some point deeper down, complexity starts to decline as the number of ascendants declines dramatically and the ‘leaves’ are reached (these classes have a high DIT). To summarize, we suggest that classes in the middle of an inheritance hierarchy may be first class citizens and need to be

changed on a regular basis due to pressure from classes both above and below them. This pressure is not as acute on classes either side of the interval.

For NOC on the other hand, the demands placed on a super class by many immediate children may mean that the super class has to be changed in response to the ever-changing requirements of those subclasses. One might expect a class with many children to be changed relatively often. It is fairly counter-intuitive to report therefore that a class with > 20 children is less change-prone than classes with between 10 and 20 children. We suggest that with > 20 children, the developer becomes less inclined to make changes to the parent class than for classes with 10-20 children.

5 Fault analysis

During the one year period over which faults were collected, 776 changes were made to classes to resolve identified faults in the WebCSC system. 495 of the 7,439 classes had fault fixes applied to them over the period studied. A large number of these classes (346) had only one fault fix applied during the period, followed by 90, 34 and 12 for two, three and four fixes being applied respectively. The highest number of fault fixes applied to a single class was 14. Table 5 shows the breakdown of number of changes made to resolve a fault per class. The mean number of faults per class across the whole system was 0.10.

Table 5 - Number of faults per class

No. faults	1	2	3	4	5	6	7	8	9	10	11	12	13	14
No. classes	346	90	34	12	6	2	0	0	1	1	0	1	1	1

5.1 Hypotheses H4-H6

In common with the analysis of changes, we explore the trends in faults for the WebCSC system through three hypotheses.

Hypothesis H4: Are larger classes more fault prone? A larger class has more functionality and there is a greater likelihood that some functionality in the class will need to be repaired as a result of a fault.

Hypothesis H5: Classes located high up in an inheritance hierarchy will be more fault-prone than other classes. Such a class has more dependants and there is therefore a greater likelihood that some functionality in the class will need to be enhanced and therefore be the cause of a fault.

Hypothesis H6: Classes with a large number of children will be more fault-prone than other classes. This hypothesis is based on the belief that a class with many children will be the subject of greater maintenance activity, since there are added dependencies on the parent class because of the changing requirements of a large number of children.

5.1.1 Class size and faults (H4)

Figures 10-16 show the graphs of the correlations between class size features and faults. Table 6 shows the correlation coefficients for each of those Figures. The Pearson correlation coefficient shows that all measures except for static methods were statistically significant against fault proneness, while the Spearman rank correlation coefficients show that all measures are correlated significantly with faults, with the number of static methods being the weakest.

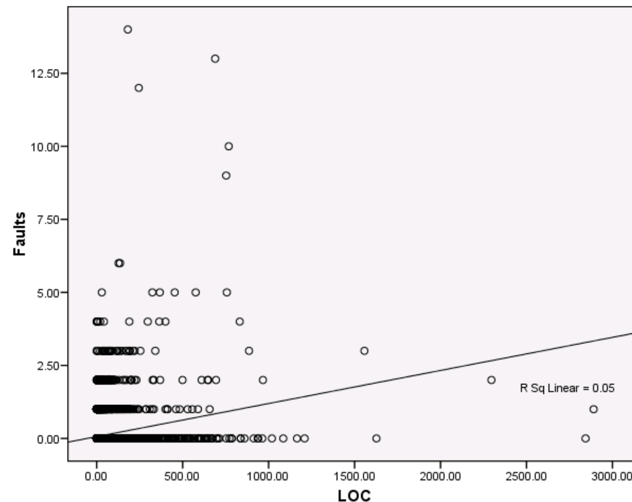

Figure 10 - LOC vs. number of faults

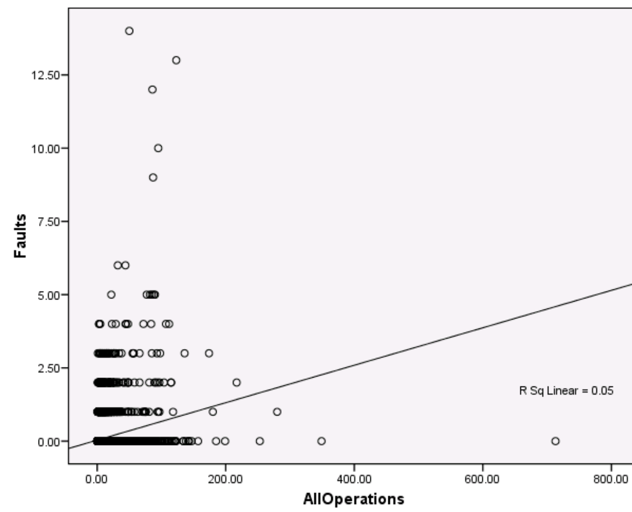

Figure 11 - All operations vs. faults

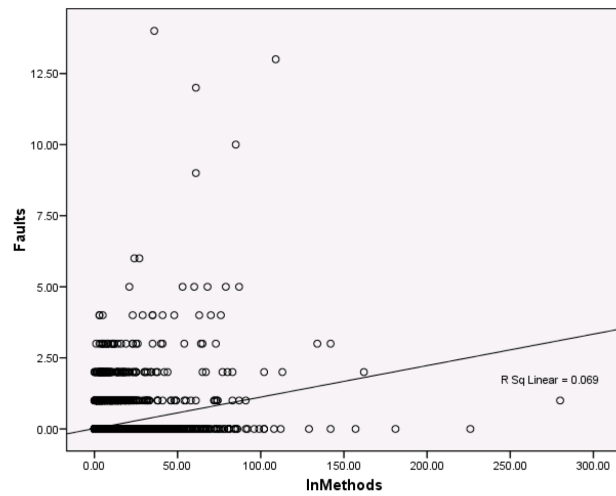

Figure 12 - Instance methods vs. faults

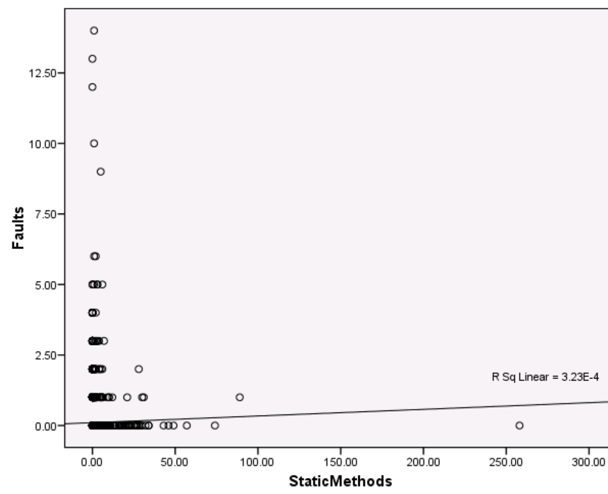

Figure 13 - Static methods vs. faults

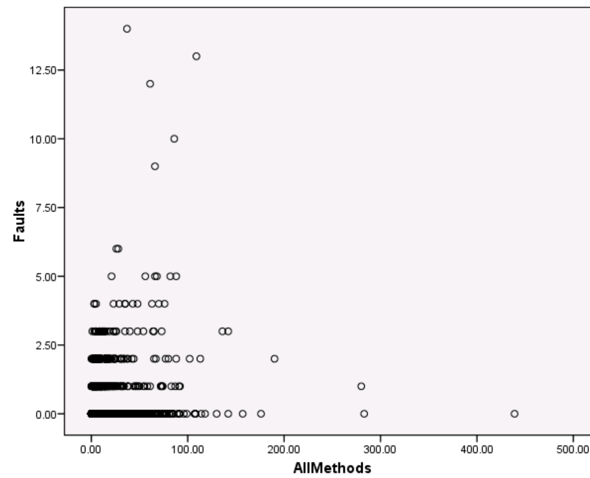

Figure 14 - All methods vs. faults

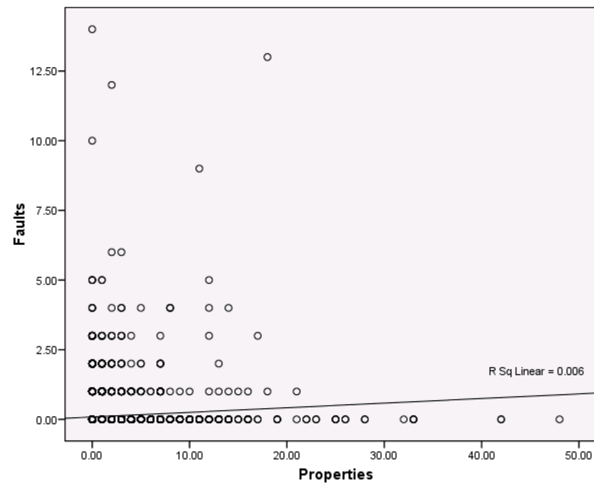

Figure 15 - Properties vs. faults

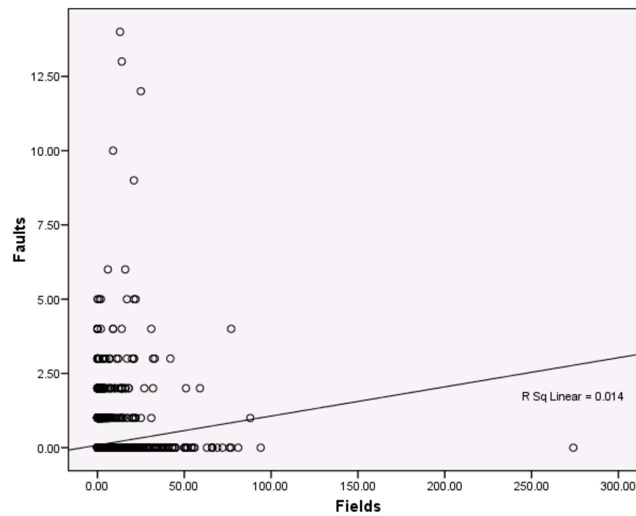

Figure 16 - Fields vs. faults

Table 6 - Correlation coefficients (class features versus faults)

Metric	Pearson's	Spearman's
LOC	0.22*	0.16*
Operations	0.22*	0.16*
Instance methods	0.26*	0.16*
Static methods	0.02	0.04*
Methods	0.24*	0.17*

Properties	0.08*	0.07*
Fields	0.12*	0.07*

We can clearly find support for our original hypothesis H4 that large classes are more fault-prone. However, as we found for static methods (and interestingly the same trend as shown in Table 2), there are some class features that did not show as strong a correlation. In other words, the propensity of faults is not as great for static methods. We also note from Table 6 that fields are significant at the 1% level for both coefficients, but this trend is not reported in the values for fields in Table 2. To determine whether influence of inheritance characteristics on fault propensity we used the same two inheritance-based metrics as used previously, namely DIT and NOC.

5.1.2 DIT analysis (H5)

Table 7 shows the number of classes and the mean number of faults per class grouped by DIT. Figure 17 clearly shows that once DIT becomes greater than 3, classes become more prone to faults; the evidence here again seems to support that trend. The interval effect is again clear between DIT 4 and 6 inclusive.

Table 7 - Classes and mean number of faults per class grouped by DIT

DIT	Number classes	Mean No. faults
0	1116	0.088
1	2445	0.090
2	1836	0.108
3	1362	0.065
4	296	0.193
5	92	0.272
6	41	0.585
7	214	0.192
8	52	0.058
9	31	0.645
10	1	0.000
11	1	0.000

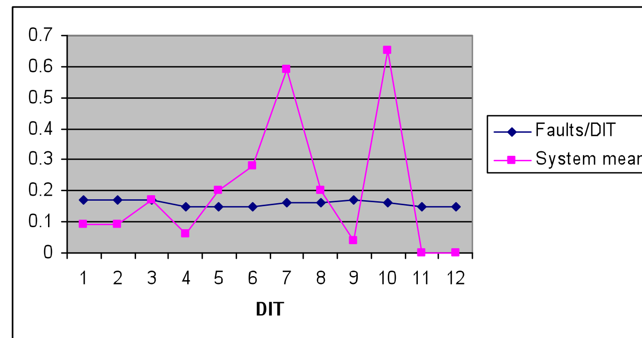

Figure 17 - Mean faults per DIT

We find little support for hypothesis H5 from the available data. It is not true that classes with a high DIT are more fault-prone than other classes. In keeping with the results so far, it is in the middle tiers of the inheritance hierarchy that faults tend to occur.

One noteworthy observation for the analysis so far is that classes at deep levels tend to be the most change-prone and fault-prone. A recent study by Nasseri et al. [Nasseri08] reported that 96% of incremental class changes over the course of the versions of four Java open-source systems 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 below; this was largely because the majority of the system's classes were at DIT 1 and 2. It would appear that maintaining shallow DIT levels might be one policy that developers adopt to avoid the problems that we see emerging for the WebCSC system. That is not to say, of course, that there is conclusive proof that a shallow inheritance hierarchy is any better in terms of fault propensity. For open-source systems however, it seems to be a common policy to follow.

The nature of open-source with geographically disparate developers, who may not be aware of the overall system design, may be the root cause of very shallow inheritance hierarchies. If a developer is not familiar with the overall inheritance hierarchy, then that might inhibit certain changes being made to the same hierarchy by that developer.

5.1.3 NOC analysis (H6)

The NOC values collected showed that the majority of classes had zero subclasses. Those classes had a similar rate of faults to the mean of the entire system (0.10 faults per class). Classes with an NOC between 1 and 10 however, had a higher rate of faults than the mean of the system. This increased for classes with an NOC between 11 and 20 and again for classes with an NOC above 20. The NOC

measurements clearly show a trend for classes with a higher number of children to have an increased propensity for faults. Table 8 shows the number of classes and the mean number of faults per class grouped by the number of children. The striking feature of Table 8 is the jump from NOC 1-10 to NOC 11-20 (the mean faults almost double in the transition). It is interesting that there is no interval effect in evidence as there is for inheritance depth or for NOC changes.

Table 8 - NOC, number of classes and mean number of faults

NOC	Classes	Mean faults
0	5951	0.10
1-10	1412	0.13
11-20	75	0.23
>20	49	0.25

In terms of our original hypotheses H6, we conclude that classes with a large number of children are more fault-prone than those classes in a simpler inheritance hierarchy. Since the values in Table 8 show that classes with greater numbers of children were more likely to be changed, it implies that the number of faults in those classes are likely to be correspondingly fault-prone. One lesson that we can learn from our analysis is that restricting the number of children *per se* can go some way to limiting the number of changes likely to be made to a class and potentially the number of faults.

6 Discussion

The study presented raises a number of issues for the developer and a strategy for minimizing a) the likely changes that need to be made to a class and b) the faults that arise from a class.

First, the argument in favor of limiting the depth to which an inheritance hierarchy should grow results in a dilemma - restricting inheritance depth will inevitably cause inheritance width to grow to compensate – and we have shown in the previous section how that can also be problematic. It would seem that, from our study, the size of the entire inheritance hierarchy (both in depth and width) should perhaps be restricted so that depth and width are moderated. Our data suggests DIT 3 to be the threshold value and NOC to be limited to as small as possible a value. Second, the study highlights a) the need to maintain a pragmatic view of the entire hierarchy and b) the vigilance needed on the part of developers and project managers to apply consistent, remedial techniques such as refactoring [Fowler99] and re-engineering; for example, the replacement of inheritance with

aggregation or other forms of coupling [Johnson93]. Third, if, as we suggest, there is a level of DIT and NOC which show a higher propensity for faults, then what is a project manager or developer to do in the face of consistent pressure for a system to grow in size as it evolves [Girba05]? We could propose that a code smell analysis [Fowler99] could be used to determine the point at which re-engineering and/or refactoring should take place and, in that sense, the warning signs can be highlighted. Equally, there may be a case for amalgamating classes or even collapsing a hierarchy to avoid its depth becoming too large.

For a study of this type, the threats to its validity also need to be considered [Fenton97]. First, we have to consider that only one system was used as a basis of the study. However, we feel we are adding to the knowledge already accumulated by the two previous studies of Bieman et al. and, in that sense, our work is a contribution. Second, we have used a C# system and previous studies have used a combination of C++ and Java only. In defense of this threat, the differences between C# and Java are relatively minimal. We feel that our study actually adds to our knowledge of trends in different OO languages. Third, since we have collected faults and change data, one criticism is that they are likely to produce the same results anyway since most faults induce the requirement to make code changes. However, the period in which we studied the faults for the WebCSC system was for a shorter period than that for changes. Also, the fault data represented only a very small part of the overall changes made to the system. Finally, we have assumed that each 'change' and 'fault' are equivalent in nature, when in actuality there would be large differences in the size of a) each change made and b) severity of each fault fixed. Future work could consider a form or normalization for change size and/or fault to determine if any different results became evident.

7 Conclusions and future work

In this paper, we have described an evolutionary study of change extracted from a large commercial C# system [Kemerer99]. Change was measured against the design context of the classes within the system, more specifically size and inheritance characteristics. Results showed a strong positive correlation between the class size measures and change-proneness but this was not true for class features studied. Classes within a specific range of inheritance depth and number of children were found to be relatively more prone to change - the fault data showed similar results. The most striking result to emerge was the notion of an inheritance depth 'interval' between which change and fault-proneness were at their highest. Below and above that interval, however both features were less acute.

One question that arises from this study is how the results could inform developer practice. The first issue is that if large classes are changed more often, then would decomposing large classes through, for example, refactoring actually reduce the total number of changes? One view would be that relatively small classes are more cohesive and hence while this might not mean a reduction in the number of changes, any necessary changes to smaller classes are likely to be more efficiently achieved. In other words, active re-engineering might pay dividends at a later stage. Finally, from an inheritance perspective, it would seem that extending the hierarchy beyond a certain level might be the cause of significant extra maintenance activity. Most evidence suggests very shallow inheritance hierarchies in OO systems and the evidence presented in this paper suggests that, counter-intuitively, restricting inheritance depth might be a sensible strategy.

In terms of future work, we intend to sub-categorize the changes made to the WebCSC system to determine those that are actually refactorings as opposed to regular, 'other' maintenance changes [Demeyer00]. The authors have already extracted a set of fifteen refactorings from the same system and this is described in [Gatrell09a]. We would then be in a position to determine the relationship between the fault data used in this study and the same set of refactorings. Second, we would like to form a link between design pattern-based classes and the same fault data [Gatrell09b]. This would extend the earlier studies of Bieman et al. [Bieman01, Bieman03] which only looked at changes to patterns, rather than faults therein. Finally, the WebCSC system is an ever-developing artifact. From an evolutionary perspective it would be interesting to observe whether the same trends recur as the system ages further. The study therefore represents an ongoing snapshot of the WebCSC system rather than a definitive study.

References

[Arisholm06] 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.

[Basili96] Basili, V.R., Briand, L.C. and Melo, W.L., A validation of object-oriented design metrics as quality indicators, *IEEE Trans. on Soft. Eng*, 22(10), pp. 751-761, 1996.

[Bieman95] 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.

[Bieman03] Bieman, J., Straw, G., Wang, H., Munger, P., Alexander, R., Design patterns and change proneness: an examination of five evolving

systems. *Proc. 9th International Software Metrics Symposium* (Metrics 2003), pages 40-49, 2003.

[Bieman01] Bieman, J., Jain, D., and Yang, H., Design patterns, design structure, and program changes: an industrial case study. *Proceedings. IEEE Conf. on Software Maintenance* (ICSM 2001). November 2001, Florence, Italy, pages 580-.

[Cartwright00] 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.

[Chidamber94] Chidamber, S. R., and Kemerer, C. F., A metrics suite for object oriented design, *IEEE Transactions on Software Engineering*, vol 20, no.6. pp. 467-493, 1994.

[Daly96] Daly, J., Brooks, A., Miller, J., Roper, M., Wood, M., An Empirical Study Evaluating Depth of Inheritance on the Maintainability of Object-Oriented Software, *Empirical Soft. Eng., An Intl Journal*, 1(2):109-132, 1996.

[Demeyer00] Demeyer, S., Ducasse, S., and Nierstrasz, O., Finding refactorings via change metrics, *ACM Conference on Object Oriented Programming Systems Languages and Applications* (OOPSLA), Minneapolis, USA, 166-177, 2000.

[Fenton97] Fenton, N., and Pfleeger. S., Software Metrics - A Rigorous and Practical Approach Second Edition. Int. Thompson Computer Press, London, 1997.

[Field05] Field, A.: Discovering Statistics Using SPSS (Sage Publications, 2005)

[Fowler99] Fowler, M., Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[Gatrell09a] Gatrell, M., Counsell, S., and Hall, T., Empirical Support for Two Refactoring Studies Using Commercial C# Software, *Proceedings of Empirical Assessment in Software Engineering* (EASE 2009), Durham, April 2009.

[Gatrell09b] Gatrell, M., Counsell, S., and Hall T., Design patterns and change proneness: a replication using proprietary C# software, *Proceedings of the IEEE Working Conference on Reverse Engineering* (WCRE'09), Lille, France, October, 2009.

[Girba05] 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.

[Gorschek10] Gorschek, T., Tempero, E., Angelis, L.. A large-scale empirical study of practitioners' use of object-oriented concepts, *Proceedings of the*

32nd IEEE/ACM International Conference on Software Engineering, Cape Town, South Africa, 115-124.

[Harrison97] Harrison, R., and Counsell, S., and Nithi, R., Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented Systems, *Journal of Systems and Software*, 52/2-3, June 2000, pp. 173-179

[Johnson93] Johnson, R., and Opdyke, W., Refactoring and aggregation, In Object Technologies for Advanced Software, *First Japan Society for Software Science and Technology (JSSST) International Symposium*, volume 742 of Lecture Notes in Computer Science, pages 264-278, 1993.

[Kemerer99] Kemerer, C. and Slaughter, S., Need for more Longitudinal Studies of Software Maintenance, *Empirical Soft Engineering: An Intl. Journal,* 2(2), pages 109-118, 1999.

[Nasseri08] 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.

[Prechelt03] 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.

[Wood99] 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.

About the author(s)

Matt Gatrell is the Director of Development and Chief Architect for an R&D programme at an international software company specialising in Transaction Content Processing. He is currently completing his PhD at Brunel University. His research interests include empirical software engineering, design patterns, refactoring and testing.

Steve Counsell received his BSc in Computing in 1987, an MSc. in Systems Analysis in 1988 and a PhD in 2002. From 1998-2004, he was a Lecturer in Computer Science at Birkbeck, London. He is currently a Lecturer in the School of Information Systems, Computing and Mathematics at Brunel University.