An Empirical Investigation of an Object-Oriented Software System

Michelle Cartwright and Martin Shepperd
Department of Computing
Bournemouth University
Talbot Campus
Poole, BH12 5BB
England
{mcartwri, mshepper }@bmth.ac.uk

Abstract

This paper describes an empirical investigation into an industrial object-oriented (OO) system comprising 133,000 lines of C++. The system was a sub system of a telecommunications product and was developed using the Shlaer-Mellor method. From this study we found that there was little use of OO constructs such as inheritance and therefore polymorphism. It was also found that there was a significant difference in the defect densities between those classes that participated in inheritance structures and those that did not, with the former being approximately three times more defect prone. We were able to construct useful prediction systems for size and number of defects based upon simple counts such as the number of states and events per class. Although these prediction systems are only likely to have local significance, there is a more general principle that software developers can consider building their own local prediction systems. Moreover, we believe this is possible even in the absence of the suites of metrics that have been advocated by researchers into OO technology. As a consequence, measurement technology may be accessible to a wider group of potential users.

Keywords: metrics, object orientation, empirical analysis.

1. Aims of the Investigation

Although the original ideas behind object oriented technology (OOT), derive from work on the programming language Simula in the 1960s, it was not until the 1980s that the work was popularised and its use became more widespread. Presently, C++ and Java are widely used and widely taught. The OO paradigm could be regarded as the orthodoxy of the late 1990s. Unfortunately, with a few notable exceptions such as [5, 9-11, 17], we have comparatively little empirically based knowledge of the behaviour of systems that have been implemented using OOT. Thus as OOT, and

particularly the use of C++, continues to be heavily invested in, the need for research into better understanding, and prediction, of the behaviour of OOT is becoming a matter of some urgency.

Despite the need for empirical research into large-scale OO systems, the majority of object-oriented metrics research has concentrated upon defining sets of structural metrics, e.g. [1, 6]. Although this can be a useful precursor to empirical work, on its own this type of approach has limited practical utility. Without empirical evidence it is not possible to say how effective these measures are, particularly in the sense of being inputs to prediction systems (e.g. of defects, reliability, cost etc.) that are able to yield engineering approximations to aid in the process of developing software. Moreover, this type of analysis yields somewhat indirect insights into the technology itself [7].

The objectives of our investigation — based upon a significant piece of industrial OO software — are twofold, first to contribute to our understanding OO technology and second to explore the possibility of building prediction systems that are useful to practising software engineers. Clearly there is an interaction between these two aims.

The remainder of this paper briefly outlines the background to our empirical investigation and provides a brief account of the development method employed, namely Shlaer-Mellor. This is followed by a description of the data collection undertaken and the analysis carried out, particularly of defect distributions. There follows a discussion of the problem of building prediction systems and an evaluation of some simple local models to predict defects and size. The paper concludes by relating our results to those deriving from other empirical studies.

2. System Background

This investigation was based at a large European telecommunications company. Presently, the organisation employs approximately 20,000 staff and more than 2,000 software developers. A disciplined approach is adopted for software design and implementation and the company is ISO 9000 accredited. There is considerable emphasis upon software quality, in particular eliminating defects. As much as 45% of resources are devoted to testing and simulation. To that end complex testing environments have been developed. The company places a high value on training and staff development and has an interest in new and innovative software development methods and techniques.

The system studied is a sub-system of a much larger industrial real-time telecommunications system which comprises several million lines of code (LOC) and has been evolving over the past ten years. Its success is central to the organisation's financial health. The sub-system is written in C++ and has been designed using the Shlaer-Mellor method. It consists of 32 classes (or Shlaer-Mellor objects) which corresponds to slightly over 133,000 LOC. The sub-system was intended to add new functionality to the existing product. Design documentation, incident reports and change data

from a version control system were made available. The system has been delivered and the change data supplied refers to defects raised at integration testing and post delivery. All other changes, such as those made for enhancement purposes, have been excluded. At the time of the analysis the system had been in operation for approximately 12 months.

The developers were experienced software developers, the average level of experience being in excess of ten years. They also had substantial experience of telecommunications domain, although the particular application was relatively novel. This was the first use of OOT by the team, consequently, members had attended a series of training courses both for OO analysis and in C++. A minority of the team had had previous practical experience of C++.

We now briefly describe the Shlaer-Mellor method since it is not as widespread as some other methods such as Booch [4] and Rumbaugh [14]. Shlaer-Mellor presents itself as an analysis method, but in practice, certainly in this case, it covers high level design. The method is aimed at real-time projects and is relatively mature, originating from 1979, with books being published in 1988 [15] and 1992 [16]. Its origins from the structured design and analysis methods of the 1970s can be traced quite clearly.

The method consists of three main models, namely information, state and process models. The *information model* is used to identify objects (these are actually classes in the more widely accepted terminology), their attributes and the relationships between objects. The *state model* is used to catalogue the behaviour of objects and relationships over time, using state transition diagrams and tables. The *process model* takes the actions of the state models and defines them in terms of processes and object data stores (these correspond to the data of an object on the information model). Each action is depicted as an action data flow diagram, using the notation of a traditional data flow diagram.

A system is divided into domains, each relating to a distinct area or subject of the whole system. There are four classes of domain. The *application domain*, normally one per system, is the subject matter from the point of view of the user, (i.e. what the system is intended to do). The *service domain* provides mechanisms and utilities to support the application domain. Examples of the service domain are the user interface, input/output and data recording/archiving. The *architectural domain* provides mechanisms and structures to control the system and for data management. It also provides policies for uniformity of software construction, such as specifying how data is to be organised and accessed. This domain will provide system support activities, e.g. initialisation and shutdown, or switchover to a standby system. This domain may also be concerned with portability issues and with performance measurement of the system once it has been implemented. Finally, *the implementation domain*, is concerned with how the requirements (of the architectural domain) are implemented, using the prescribed programming languages, operating systems, class libraries and networks.

Domains are linked via *bridges*, to allow one domain (client) to make use of the services or mechanisms of another (server). Domains can themselves be divided into sub-systems, partitioned according to the clusters into which groups of objects (classes) fall on the information model. For further details of the method, and the processes involved, the interested reader is referred to Project Technology's web site at http://www.projtech.com/.

3. Data Collection

The software developers utilised the version control system SCCS from integration testing onwards. The organisation also maintained a database of all incident reports from system testing onwards. Data from these sources were collected and analysed in order to trace defects to specific file changes. Fortunately each class was implemented as an individual file so that it was possible to trace defects to classes. The distribution of defects could then be used for project planning and to help extend our understanding of OO software in industry.

Initially we had considered collecting the Chidamber and Kemerer (CK) metrics suite [6]. Unfortunately only two out of the six metrics were readily available from the available design documentation. These were DIT (depth of inheritance tree) and NOC (number of children). Consequently, we decided to supplement these metrics with a number of additional measures that could be easily collected at the analysis/design stage. The sources utilised were the CASE tool (Teamwork) model consisting of an information model and state charts, code statistics and defect data.

Mnemonic	Variable	Explanation
ATTRIB	Attributes	Count of attributes per class from the information model.
STATES	States	Count of states per class in the state model
EVNT	Events	Count of events per class in the state model
READS	Reads	Count of all read accesses by a class contained in the CASE tool.
WRITES	Writes	Count of all write accesses by a class contained in the CASE tool.
DELS	Deletes	Count of all delete accesses by a class contained in the CASE tool.
RWD	Read/write/ deletes	Count of synchronous accesses (i.e. the sum of READS, WRITES and DELS) per class from the CASE tool.
DIT	Depth Inheritance Tree	Depth of a class in the inheritance tree where the root class is zero.
NOC	Number of	Number of child classes.

	Children	
LOC	Lines of code	C++ lines of code per class.
LOC_B	Lines of code (body)	C++ body file lines of code per class.
LOC_H	Lines of code (header)	C++ header file lines of code per class.
DFCT	Defects	Count of defect per class.

Table 1: Variables Collected

Table 1 lists the variables collected. The first nine variables characterise the OO system architecture or structure and these may be collected at analysis or design time. Note that duplicates are eliminated from the counts of events and synchronous accesses. The remaining four metrics are external measures that might be of interest to project managers. Ideally we would, in addition, have collected effort data. Unfortunately effort data by class proved to be unobtainable. However, the LOC information could be regarded as a crude proxy. Note that LOC is counted as the number of end of line markers, ";").

4. Data Analysis

This section first considers the system as a whole and then proceeds to analyse the data on a class by class basis.

4.1 Analysis of the System

The system comprised of just over 133 KLOC of C++. Table 2 provides a more detailed breakdown.

Total LOC	133 632
Body files	109 603
Header file	24 029
Classes	32
Defects	259

Table 2: System Data

Using the data from Table 2 we can derive an overall defect density of 1.94 $KLOC^{-1}$ which compares quite favourably with defect levels quoted by Hatton [11] of 2.9 $KLOC^{-1}$. Indeed, the figure is quite conservative since some of the defects that have been recorded were found after integration testing but prior to release.

There were just two inheritance trees or structures in the system, one of two levels consisting of seven classes and the other of one level, consisting

of five classes[1]. There are two possible explanations for this. Firstly it may be that there is little in the problem domain that naturally lends itself to inheritance. Secondly, and alternatively, the analysis and design method used, Shlaer-Mellor, does not provide explicit support for inheritance — it is not discouraged, but there is no guidance of how to look for possible inheritance hierarchies as there is in some other OO methods. As a result of discussions with the developers it became apparent that they endeavoured to avoid the use of inheritance since they were of the opinion that it would be harder to understand and therefore maintain. The lack of use of the inheritance mechanism is in line with other researchers' findings, see for example [5].

4.2 Analysis by Class

This section considers the data on a class by class basis. The raw data may be found in Appendix A.

Variable	Mean	Median	Min	Max
ATTRIB	8.66	4.5	1	32
STATES	18.03	13	0	114
EVNT	20.53	10.5	0	122
READS	16.25	11.5	0	83
WRITES	14.22	8.5	0	56
DELS	1.50	1	0	5
RWD	31.97	22	0	131
DIT	0.44	0	0	2
NOC	0.31	0	0	4
LOC	4178.50	3524.5	603	20165
LOC_B	3427.59	2775.5	396	17177
LOC_H	750.91	707	207	2988
DFCT	8.09	2	0	47

Table 3: Summary Statistics of Variables Collected

Table 3 shows some basic summary statistics in the form of the mean, median, minimum and maximum value for each variable collected. The first nine metrics are internal metrics whilst the next four are external

[1] The first level or root of an inheritance tree is counted as level 0, with its immediate subclasses as level 1 and so on.

metrics that may be of management interest. It is apparent that since the median value is in all cases lower than the mean each variable exhibits some tendency to skew positively. This is the consequence of a few very large classes.

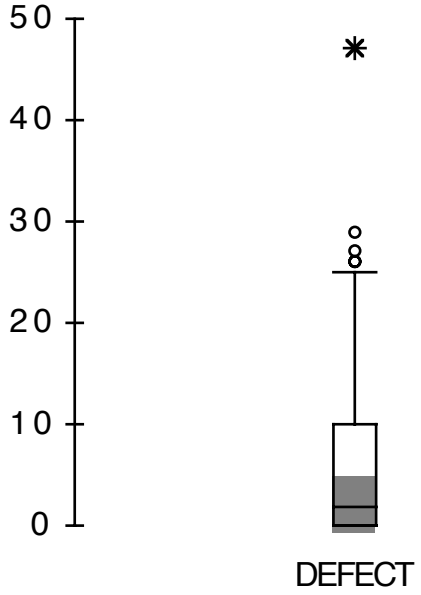

Figure 1a: Boxplots of Defects per Class

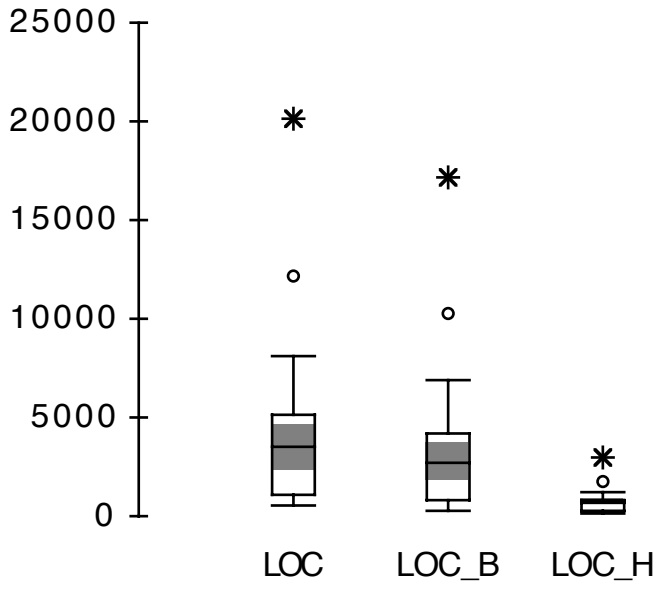

Figure 1b: Boxplots of LOC per Class

Figures 1a and 1b reveal the skewed nature of the distributions of some of the metrics. Note that 'o' represents an outlier and '*' an extreme outlier. Figure 1a indicates a small number of very defect prone, and indeed a

mere 22% of the classes account for 75% of all defects, more evidence of a 20:80 rule as reported by others, for [6, 12]. Figure 1b indicates several unusually large classes, one in excess of 20000 LOC.

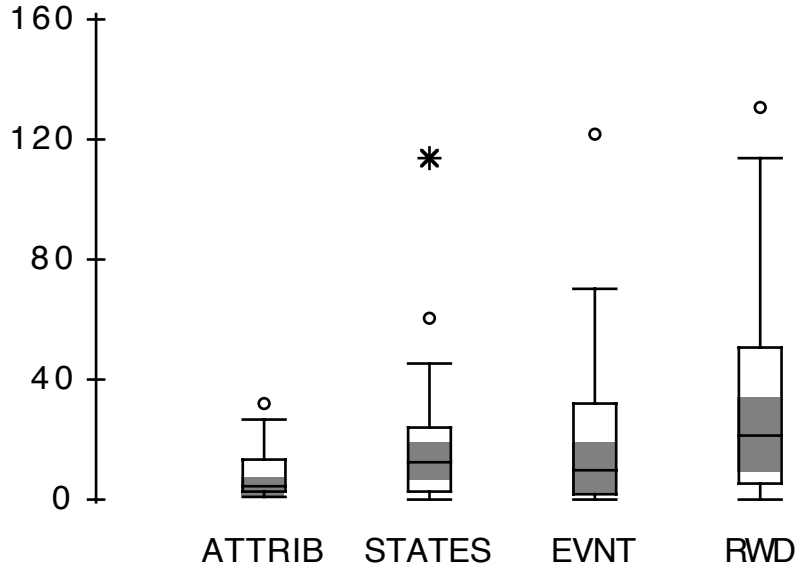

Figure 1c: Boxplots of Architectural Metrics

Figure 1c shows the distribution of values for some of the architectural or design metrics. Again there are a number of outliers. This skewing indicates the need to utilise parametric tests cautiously and to beware of the effect of a small number of outlier values.

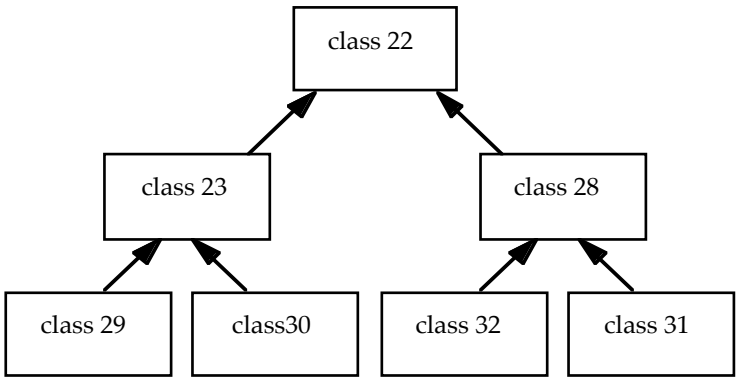

Fig. 2: Inheritance hierarchy containing the outlier classes

Class 22 is by far the largest class in the system (114 possible states and LOC = 20165, compared with the next largest, class 23, with 60 possible states and LOC 12101 and with median class size of 13 possible states and LOC of 3524.5). It would appear that many of the measures taken are size driven. More interesting then is to see that both of these classes are part of the same class inheritance hierarchy (see Figure 2).

	ATTRIB	STATES	EVNT	RWD	LOC	DFCT
ATTRIB	1.000					
STATES	0.562*	1.000				
EVNT	0.318*	0.898*	1.000			
RWD	0.508*	0.858*	0.859*	1.000		
LOC	0.563*	0.968*	0.910*	0.848*	1.000	
DFCT	0.166	0.751*	0.838*	0.769*	0.759*	1.000

* = significant at 5% confidence level.

Table 4: Spearman Rank Correlations

Table 4 contains the results of a Spearman cross correlation of some of the variables collected (the full cross correlation is to be found in Appendix B). Note that DIT was not included since it only took on three values (0, 1 or 2) in this dataset, likewise NOC also with three values (0, 2 or 4). A non-parametric test was used due to possible problems of outliers and skewed distributions. All correlation coefficients, but one, are significant at the 5% confidence level, in other words all coefficients other than ATTRIB against DFCT. Clearly there is considerable inter item correlation. For example, all variables are significantly correlated with LOC — many very strongly — which suggests that variables such as STATES are proxies for size. It also suggests that a size effect may dominate, that is, as classes become larger so they contain more attributes, states, synchronous accesses and become more defect prone. For this reason the measures were size normalised in order to look for effects which might otherwise be swamped by size. Such a size normalisation procedure is a relatively commonplace procedure amongst metrics researchers, see for example [8, 13].

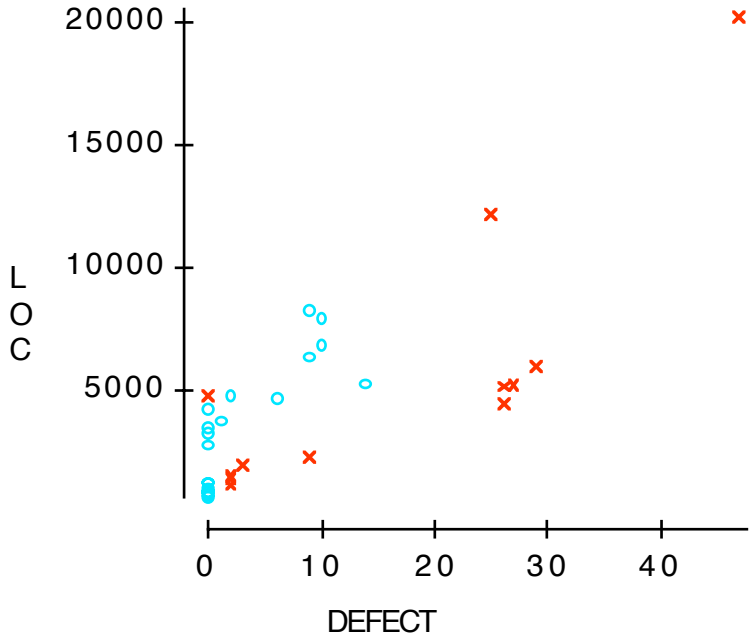

Figure 3: Scatterplot of LOC against Defects

Figure 3 uses an 'X' to represent classes that participate in an inheritance structure and an 'O' to represent singleton classes. It was suspected that density of defects would be higher for classes in an inheritance structure than for those not involved in an inheritance structure.

Group	Count	Mean	Median	Min	Max
No inheritance	20	3.05	0	0	14
Inheritance	12	16.50	17	0	47

Table 5a: Defects by Class

Group	Count	Mean[1]	Median	Min	Max
No inheritance	20	0.90	0	0	2.70
Inheritance	12	3.01	2.20	0	5.85

Table 5b: Defect Densities by Class

Table 5a shows a range of descriptive statistics for defects per class divided into two categories, those not participating in inheritance structures and those that do. Likewise Table 5b compares these categories of class this time based upon defect density. Here the data reveals means of 0.89 defects

[1] Using a mean of means would yield 0.5 and 2.97 defects per KLOC, however, this is somewhat misleading due to the variation in class size.

per KLOC and 3.01 defects per KLOC respectively. An unpaired two tailed t-test was applied to assess whether those classes involved in inheritance structures were truly from a distinct sub-population, or whether the apparent increase in defects for inheritance classes occurred by chance. The result confirmed that they were indeed from a distinct sub-population, the F-value being calculated at 6.33, compared with a tabled value of 4.17 ($p<0.001$). Additionally the highest defect densities calculated were for classes at the bottom of their respective inheritance hierarchies.

5. Building Prediction Systems

The next step was to assess the possibility of building prediction systems f nal attributes of defect proneness (DFCT) and size (LOC). The equations presented below have been selected on the basis that they are simple, containing one or at most two variables. Simple models are preferable to more complicated alternatives since they involve less effort, not only in calculation, but more importantly in metrics collection. They also tend to be more robust since they have fewer problems of collinearity. This is quite a significant consideration since we have already uncovered potential problems of high levels of inter item dependency (see Table 4).

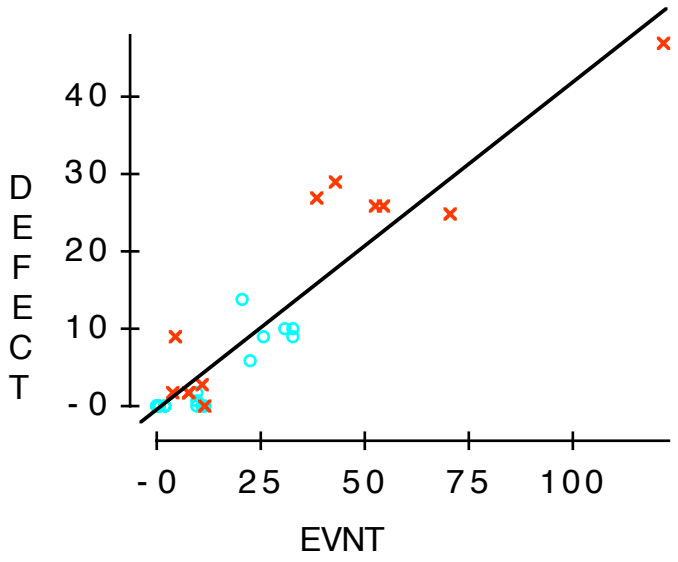

Figure 4: Scatterplot of Events against Defects

First we consider defects. Fig. 4 shows the relationship between the number of events per class in the Shlaer-Mellor state model and the number of subsequent defects to the class. An 'x' denotes a class in an inheritance structure and an 'o' a class that has neither parents nor children classes. A regression line is fitted which can be seen to be a good fit as evidenced by the high R squared (see below).

```
R squared = 87.6%      R squared (adjusted) = 87.2%
```

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	3821.50	1	3821.50	213
Residual	539.22	30	17.97	

Variable	Coefficient	s.e. of Coeff	t-ratio	prob
Constant	-0.58	0.9566	-0.602	0.5520
EVNT	0.42	0.0290	14.6	≤ 0.0001

This analysis suggests that it is possible to predict 87% of the variation in DFCT in terms of a simple equation based upon a near zero intercept (-0.58) and a slope of 0.42*EVNT. Whilst the constant is slightly negative — a somewhat counterintuitive finding — with p=0.552, it is not actually significantly different from zero. Such a simple equation is unlikely to suffer from the problems of over fitting which can afflict multiple regression analysis.

Finally, we add a dummy variable INHRTS (0 if DIT=0 or NOC=0, else 1) to the regression equation to explore whether the use of inheritance as any additional explanatory over and above the size related independent variable EVNT.

R squared = 89.7% R squared (adjusted) = 89.0%

s = 3.941 with 32 - 3 = 29 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	3910.37	2	1955.19	126
Residual	450.35	29	15.53	

Variable	Coefficient	s.e. of Coeff	t-ratio	prob
Constant	-1.34	0.945	-1.42	0.1658
EVNT	0.39	0.030	12.8	≤ 0.0001
INHRTS	3.88	1.621	2.39	0.0234

As the analysis above indicates, both EVNT and INHRTS are significant at α=0.05. This again supports the hypothesis that classes in inheritance structures are more defect prone, even taking into account their size. We would not particularly advocate the use of this second regression equation for predictive purposes. The improvement in the adjusted R-squared is modest and the use of the dummy variable will lead to instability.

We now turn to predicting class size or LOC. This may be of interest to developers since the inputs are available at analysis time. LOC may bear relationship to effort (coding, testing and so forth). Note we would have preferred to investigate effort but unfortunately this data was unavailable.

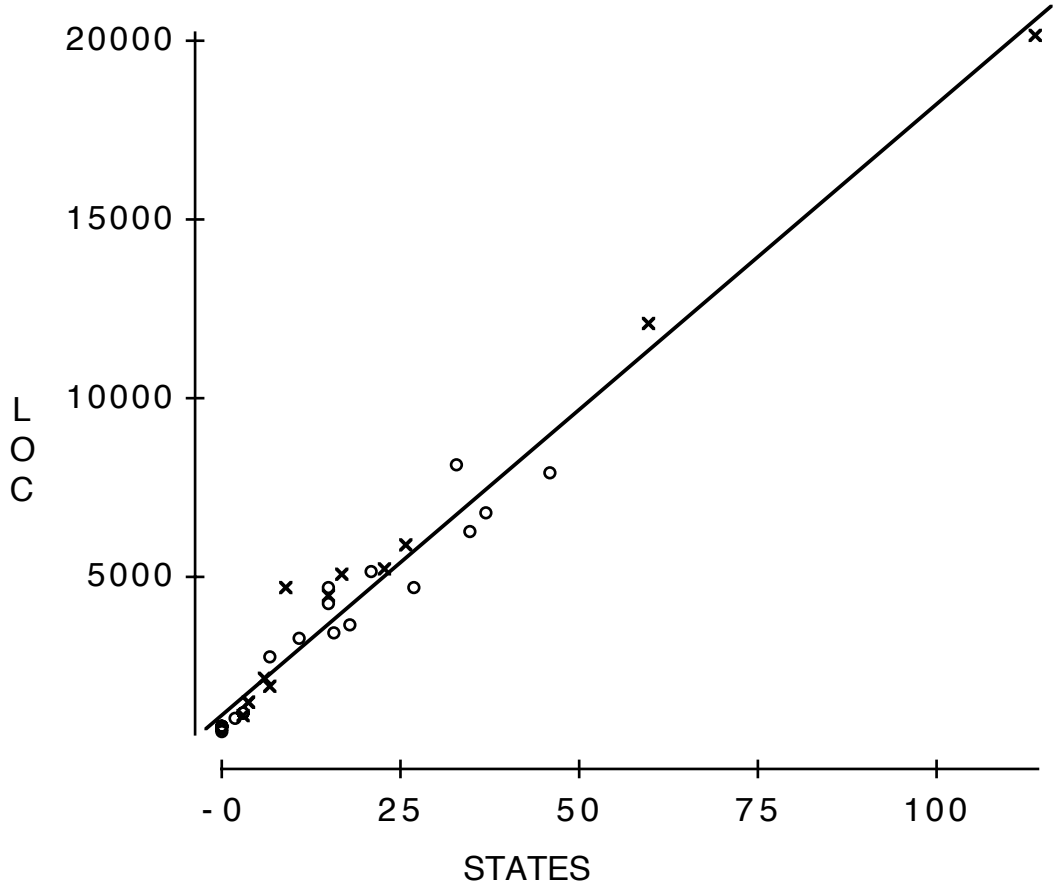

Figure 5: Scatterplot of LOC against STATES

Figure 5 shows the strong linear relationship between the size of a class as LOC and the number of states per class as contained in the state model. The regression equation intercept is just over 1100 indicating that there are some fixed overheads in terms of the size of a class, even for one which "does nothing".

R squared = 96.7% R squared (adjusted) = 96.6%

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	475082696	1	475082696	875
Residual	16296130	30	543204	

Variable	Coefficient	s.e. of Coeff	t-ratio	prob
Constant	1101.01	166.70	6.60	≤ 0.0001
STATES	170.68	5.77	29.60	≤ 0.0001

Note the very high F ratio indicating the low probability of a chance relationship and also the exceptionally high R squared indicating that over 96% of the variation in LOC can be "explained" in terms of the variable STATES. As with the prediction system for defects we again elect for a simple equation with a single independent variable.

It is well known that linear regression is vulnerable to the effect of outliers. As can be seen from the scatterplots above (Figs. 4-5) there is an outlier in the extreme top right corner of the plots.

Attribute	Median	Outlier
LOC	3524.5	20165
STATES	13.0	114

Table 6: Extreme Outlier Class Compared with Median Values

Table 6 reveals the extent to which this class is an outlier. Re-plotting without this class shows basically the same distribution and recalculating regression equations shows only very slight differences. From this we conclude that there are underlying relationships between STATES, EVNT, LOC and DFCT and that these can be exploited to build prediction systems for this environment.

The next stage was to assess the significance of the prediction systems that we had developed. A set of predicted values for LOC and DFCT were calculated from the predictive equations derived from, and compared with, the actual values to determine the residuals. The defect prediction system caused certain difficulties since we have a small number of discrete values, many of them being zero. For this reason normal accuracy indicators, such as MMRE, were not suitable[2]. We approached this problem by classifying class defect counts into four bins as follows:

Q4	> 10
Q3	4-10
Q2	1-3
Q1	0

The bins were chosen to correspond to quartiles for DFCT and predicted DFCT.

PREDICTED			ACTUAL		
	Q4	Q3	Q2	Q1	total
Q4	6	1	0	0	7
Q3	3	2	1	0	6
Q2	0	3	3	0	6
Q1	0	4	0	9	13
total	9	10	4	9	32

Table 7: Contingency Table for Predicted and Actual Defect Counts

Table 7 shows the contingency table that was generated. The chi-statistic is significant ($p \leq 0.0001$ with 9 degrees of freedom) at 38.44. This indicates

[2] To illustrate the difficulties of using MMRE to assess a defect prediction system, consider the following, where predicted $DFCT_i=1$ and actual $DFCT_i=0$. The formula for MRE_i yields $|(1-0)|/0$. Even in the event of $DFCT_i=0$ and actual $DFCT_i=1$ we still have an error of 100% whilst we would consider the prediction to be quite reasonable.

that there is a non-random relationship between actual and predicted defect counts.

By contrast, we used the MMRE indicator for the size prediction system. Here we obtained an MMRE of just under 24%. In addition, we have already noted that, both prediction systems described in this section are statistically significant ($p<0.001$), thus it is highly unlikely that the relationships occur by chance. The equations were also examined for implausible relationships, thus any equations meeting the previous criteria but intuitively implausible were rejected. Additionally all of the equations selected have a high (adjusted) R^2 value indicating that the model fits the data well — in all of the equations presented, over 80% of the variation in the dependant variable can be 'explained' by the independent variable(s) used in the equations. We therefore conclude that both prediction systems are significant. Nevertheless, this analysis has been one of model fitting which is optimistic in terms of assessing the likely accuracy of either system for making *future* predictions.

6. Conclusions

This paper has described an empirical investigation of a significant real-time C++ system that was developed using the Shlaer-Mellor method. From this study a number of findings have emerged.

First, there is very little use of many of the constructs of the OO paradigm. In particular, there is little use of class inheritance and hence polymorphism. This replicates the findings of other studies such as [5, 9]. This does not necessarily mean that the developers were "right" to avoid significant use of inheritance. Moreover, it may in part be a consequence of the development method or the fact that the C++ language does not enforce an OO approach. The fact that this was the project team's first experience of OOT may also be pertinent. They "stuck rigidly" to the Shlaer-Mellor method but this method does not explicitly support or encourage inheritance, though neither does it prevent or discourage its use. It seems likely that the caution of the team, lack of support by the method and the lack of obvious candidates for inheritance in the problem area were all factors in what would seem to be low levels of inheritance in the system. Nevertheless, inheritance as an OO mechanism caused the developers considerable concern and this may be something that the OO community wishes to address. It should be remembered that the system was built over a period 1993 to 1994, when far less guidance was available with regard to "good" or "proper" use of inheritance. Research work such as Liskov's substitution principle [18] had not permeated into standard practice and design heuristics such as [19] were not published until OO software was more established.

Second, the classes with the highest defect densities were found in the lowest levels of their respective inheritance structures. Chidamber and K DIT and NOC metrics could therefore be used to pinpoint classes that are likely to have higher defect densities. Our study indicates that the developers' caution about using inheritance has some foundation.

One question raised by this case study is whether the developers were using inheritance "properly". Qualitatively speaking we believe there may be some problems with the design (e.g. the very large class sizes and common changes to groups of classes indicating unfulfilled potential for generalisation). On the other hand one would expect any non-trivial design to be in some sense "imperfect".

This paper does not advocate that class inheritance should be avoided. However, given the higher defect densities of inheritance classes, extra consideration should be given to such classes during design reviews, code inspections and testing. We also are of the view that there is a need for further studies to assess trade-offs between different design tactics such as aggregation or inheritance.

Third, we do not believe that the unavailability of predefined sets of metrics need be a particular disadvantage for developers of OO systems wishing to adopt a quantitative approach. Our empirical work indicates that two other measures, available by the design stage, the number of events for a class (EVNT) and the number of states for a class (STATES) can be useful and accurate predictors of the number of defects and LOC. The availability of metrics will be largely determined by the methods and tools utilised by particular developers. We have shown that, for at least one OO environment, it is possible to build simple, yet useful, prediction systems for attributes such as size and defects. It is highly unlikely that the same prediction systems will fit other environments, however, the underlying principles of collecting data and developing local prediction systems are rather more likely to hold true.

The work complements other research in the field such as Basili *et al.* [3] who utilised a version of the CK metric suite modified to suit C++. Their study was based upon an experiment with student programmers. The CK metrics were extracted from the code delivered at the end of implementation, together with defect data from testing and fix data during the repair stage. The amount of modification made to a class was categorised as none, small or large, classes being allocated according to the developer's estimate of the percentage of code modified. The authors found metrics all but LCOM to be "adequate" predictors of fault prone classes. This would seem to be in line with our findings concerning the positive impact of size and inheritance upon defect levels. The major difference between the studies is that we studied a software system several orders of magnitude larger than Basili *et al.* This, however, was at the expense of control, since they were able to collect additional metrics over and above those in our case study.

By contrast, Abreu and Melo [2] have argued that inheritance is a technique to reduce the defect density in code when used "sparingly", but not at higher levels where they feel the beneficial effects will reverse. Their analysis appears to be based upon similar case studies as the Basili *et al.* work [3]. The correlation analysis suggested a negative relationship between inheritance and defects, however, their multiple regression equation contained a positive coefficient for the proportion of inherited methods. Such a position is not entirely consistent with our findings, but

may in part be explained by the scale of the artefacts we have examined and perhaps also the lack of OO experience of our developers.

To conclude, the value of this work is threefold. First, it illustrates how, using straightforward techniques based upon linear regression, it is possible to build accurate prediction systems both for size and defects. This has been achieved using a small number of measures that are all readily available early in the analysis and design stage. The prediction systems have only local significance but we believe the approach may be of wider interest. Second, we believe the patterns in the distribution of defects may enable software managers to better allocate resources. Third, the emerging body of evidence regarding the industrial application, at least of C++, suggests that like all technologies OO must be used appropriately and that there are many design trade-offs to be made, the impact of which we do not yet fully understand. The need for further published empirical studies would therefore seem to be overwhelming.

Acknowledgements

The authors would like to thank members of staff of the anonymous organisation for assistance in the data collection. We are also indebted to the referees for their detailed comments on a previous version of this paper.

References

[1] Abreu, F.B. and R. Carapuca, 'Candidate Metrics for Object-Oriented Software Within a Taxonomy Framework', *Journal of Systems and Software*, 26(1), pp87-96, 1994.

[2] Abreu, F.B. and W. Melo. 'Evaluating the Impact of Object-Oriented Design on Software Quality', in *Proc. 3rd International Software Metrics Symposium (METRICS '96)*. Berlin, Germany: IEEE, 1996.

[3] Basili, V.R., L. Briand, and W.L. Melo, A Validation of Object-Oriented Design Metrics. Technical No. CS-TR-3443, University of Maryland, 1995.

[4] Booch, G., *Object-Oriented Analysis and Design With Applications*. 2 ed. The Benjamin/Cummings Series in Object-Oriented Software Engineering, ed. G. Booch. Benjamin/Cummings: Redwood City, California, 1994.

[5] Chidamber, S.R., D.P. Darcy, and C.F. Kemerer, 'Managerial use of object oriented software metrics: an exploratory analysis', *IEEE Transactions on Software Engineering*, 24(8), pp629-639, 1998.

[6] Chidamber, S.R. and C.F. Kemerer, 'A Metrics Suite for Object-Oriented Design', *IEEE Transactions on Software Engineering*, 20(6), pp476-93, 1994.

[7] Fenton, N. and S. Pfleeger, *Software Metrics : A Rigorous and*

Practical Approach. Second Edition. 2 ed. International Thomson Computer Press: 1996.

[8] Gill, G.K. and C.F. Kemerer, 'Cyclomatic complexity density and software maintenance productivity', *IEEE Transactions on Software Engineering*, 17(12), 1991.

[9] Harrison, R., S.J. Counsell, and R.V. Nithi, 'An Evaluation of the MOOD Set of Object-Oriented Software Metrics', *IEEE Transactions on Software Engineering*, 24(6), pp491-496, 1998.

[10] Hatton, L., 'Software Failures: Follies and Fallacies', *IEE Review*, 43(2), pp49-52, 1997.

[11] Hatton, L., 'Does OO Sync with How We Think', *IEEE Software*, 15(3), pp48-54, 1998.

[12] Henderson-Sellers, B., *Object-Oriented Metrics: Measures of Complexity*. Object-Oriented Series, Prentice Hall: New Jersey, 1996.

[13] Mata-Toledo, R.A. and D.A. Gustafson, 'A factor-analysis of software complexity measures', *J. of Systems Software*, 17(3), pp267-273, 1992.

[14] Rumbaugh, J., *et al.*, *Object-Oriented Modeling and Design*. Prentice-Hall: 1991.

[15] Shlaer, S. and S.J. Mellor, *Object-Oriented Systems Analysis: Modelling the World in Data*. Prentice Hall: 1988.

[16] Shlaer, S. and S.J. Mellor, *Object Lifecycles: Modelling the World in States*. Prentice Hall: 1992.

[17] Wilde, N., P. Matthews, and R. Huitt, 'Maintaining Object-Oriented Software', *IEEE Software*, 10(Jan), pp75-80, 1993.

[18] Liskov, B., 'Data Abstraction and Hierarchy', *SIGPLAN Notices*, 23(5, pp17-34, 1988.

[19] Riel, A.J., *Object-Oriented Design Heuristics*,. . Addison- Wesley: 1996.

Appendix A: Raw Data

ATTRIB	DELS	DIT	EVNT	NOC	READS	STATES	WRITES	DEFECT	LOC	LOC_B	LOC_H
14	1	0	2	0	12	11	14	0	3213	2512	701
3	1	0	12	0	8	7	3	0	2699	2127	572
3	1	0	0	0	0	2	3	0	1041	729	312
5	1	0	2	0	0	3	5	0	1169	825	344
27	1	0	10	0	27	15	27	2	4675	3852	823
17	1	0	10	0	17	18	17	1	3655	2874	781
13	1	0	11	0	13	16	13	0	3394	2677	717
19	1	0	31	0	27	46	19	10	7946	6632	1314
5	1	0	1	0	0	3	5	0	1168	827	341
27	1	0	10	0	27	15	27	0	4198	3406	792
3	0	0	0	0	0	0	0	0	761	529	232
3	0	0	0	0	0	0	0	0	754	514	240
4	0	0	0	0	0	0	0	0	788	564	224
3	0	0	12	4	2	9	0	0	4701	3988	713
10	1	0	21	0	12	21	10	14	5181	4287	894

1	3	2	55	0	35	15	37	26	4445	3747	698
32	1	0	122	2	74	114	56	47	20165	17177	2988
5	3	2	53	0	32	17	39	26	5114	4287	827
24	0	1	71	2	83	60	31	25	12101	10320	1781
1	3	0	23	0	11	27	7	6	4630	3818	812
11	3	0	33	0	15	35	16	9	6299	5220	1079
2	2	1	8	0	9	4	5	2	1490	1119	371
2	2	1	8	0	10	4	5	2	1440	1058	382
3	2	1	5	0	3	6	6	9	2161	1652	509
2	2	1	4	0	1	3	5	2	1116	785	331
2	0	0	0	0	0	0	0	0	730	511	219
1	0	0	1	0	0	0	0	0	603	396	207
7	3	0	26	0	21	33	11	9	8155	6897	1258
16	3	0	33	0	16	37	19	10	6813	5604	1209
6	0	1	11	2	4	7	6	3	1940	1464	476
3	5	2	39	0	27	23	34	27	5239	4343	896
3	5	2	43	0	34	26	35	29	5928	4942	986

Appendix B: Rank Spearman Cross-Correlations

	ATTRIB	DELS	DIT	EVNT	NOC	READS	STATES	WRITES
ATTRIB	1.000							
DELS	-0.121	1.000						
DIT	-0.338	0.461	1.000					
EVNT	0.318	0.534	0.382	1.000				
NOC	0.258	-0.407	0.089	0.341	1.000			
READS	0.495	0.497	0.362	0.876	0.181	1.000		
STATES	0.562	0.496	0.103	0.898	0.251	0.866	1.000	
WRITES	0.528	0.568	0.383	0.813	0.093	0.944	0.821	1.000
DEFECT	0.166	0.619	0.580	0.838	0.182	0.764	0.751	0.770
LOC	0.563	0.477	0.136	0.910	0.314	0.864	0.968	0.800
LOC_B	0.563	0.479	0.140	0.911	0.314	0.866	0.968	0.802
LOC_H	0.620	0.476	0.102	0.882	0.258	0.874	0.982	0.823

	DEFECT	LOC	LOC_B	LOC_H
DEFECT	1.000			
LOC	0.759	1.000		
LOC_B	0.760	1.000	1.000	
LOC_H	0.741	0.988	0.987	1.000