An Empirical Study Evaluating Depth of Inheritance on the Maintainability of Object-Oriented Software

John Daly*, Andrew Brooks, James Miller, Marc Roper and Murray Wood
Dept. Computer Science, University of Strathclyde,
Livingstone Tower, Richmond Street, Glasgow G1 1XH, Scotland
(email: efocs@cs.strath.ac.uk)

ISERN-96-11

Abstract

This empirical research was undertaken as part of a multi-method programme of research to investigate unsupported claims made of object-oriented technology. A series of subject-based laboratory experiments, including an internal replication, tested the effect of inheritance depth on the maintainability of object-oriented software. Subjects were timed performing identical maintenance tasks on object-oriented software with a hierarchy of three levels of inheritance depth and equivalent object-based software with no inheritance. This was then replicated with more experienced subjects. In a second experiment of similar design, subjects were timed performing identical maintenance tasks on object-oriented software with a hierarchy of five levels of inheritance depth and the equivalent object-based software.

The collected data showed that subjects maintaining object-oriented software with three levels of inheritance depth performed the maintenance tasks significantly quicker than those maintaining equivalent object-based software with no inheritance. In contrast, subjects maintaining the object-oriented software with five levels of inheritance depth took longer, on average, than the subjects maintaining the equivalent object-based software (although statistical significance was not obtained). Subjects' source code solutions and debriefing questionnaires provided some evidence suggesting subjects began to experience difficulties with the deeper inheritance hierarchy

It is not at all obvious that object-oriented software is going to be more maintainable in the long run. These findings are sufficiently important that attempts to verify the results should be made by independent researchers.

*Daly is now with the Fraunhofer Institut (IESE), Kaiserslautern, Germany

1 Introduction

Object-oriented technology has become increasingly popular as a result of anecdotal evidence and expert intuition despite various warnings about relying only on such evidence [Bur95], [HHL90]. Evidence must be derived from a variety of empirical techniques, the data collected being used to substantiate findings, identify discrepancies, and act as a platform for further investigation. Unfortunately not enough of this research is being performed — for object-oriented technology this means little empirical evidence exists to support many of the claims made of it. For example, Jones [Jon94] details a visible lack of empirical data to support the assertions of substantial gains in software productivity and quality, reduction in defect potential (the probable number of defects from all causes that will be encountered during development and production) and improving defect removal efficiency (the percentage of defects removed by any operation, e.g., code inspection), and reuse of software components. Henry *et al.* [HHL90] provide a list of references which they state have made claims having qualitative appeal, but which have little supporting quantitative data.

In contrast, related research has reported that aggregation, dynamic binding, inheritance, and polymorphism can introduce difficulties for programmers attempting to understand, maintain, and test object-oriented software; see [CvM93], [Dvo94], [JKZ94], [KGH+94], [LMR92], [WH92], [WMH93]. For example, Wilde and Huitt [WH92] argue that the mechanisms of inheritance, polymorphism, and dynamic binding are responsible for the creation of delocalised plans — pieces of code that are conceptually related but are physically located in non-contiguous parts of the program [SPL+88]. As a consequence, although it can be relatively easy to understand most of the data structures and member functions individually, understanding their combined functionality can be extremely difficult [KGH+94], [LMR92]. In addition, the use of inheritance and polymorphism can create a large amount of dependencies that need to be considered within an object-oriented program [KGH+94], [WH92]. The number of dependencies that must be considered is far greater than in a conventional system and, as a consequence, a maintainer can have great difficulty identifying the impact of their changes [KGH+94].

So clearly the alleged advantages and disadvantages of the technology require substantial empirical investigation. This realisation led to a multi-method programme of research [Dal96], [DBM+95]. The programme of research began with an exploratory investigation where structured interviews were conducted, with both academics and industrialists, on their opinions of the merits and failings of the object-oriented approach. The findings of this primary investigation were used to design and implement a questionnaire on key aspects of object-oriented systems — the intention was to confirm (or otherwise) the findings of the first phase across a much larger and wider practitioner group. Finally, a series of subject-based laboratory experiments

were conducted, including an internal replication, which tested one of the important and most interesting outcomes of the questionnaire survey in a more controlled setting.

This paper details the design of these experiments and describes the procedures, subjects, tasks, and materials. Statistical tests are applied to the time data collected and these are interpreted in conjunction with an inductive analysis to explore possible explanations of the data. Finally, threats to internal and external validity are discussed.

The collected data shows that subjects maintaining object-oriented software with three levels of inheritance depth performed the maintenance tasks significantly quicker than those maintaining equivalent object-based software with no inheritance. In contrast, subjects maintaining the object-oriented software with five levels of inheritance depth took longer, on average, than the subjects maintaining object-based software (although statistical significance was not obtained). Subjects' source code solutions, and debriefing questionnaires provide some evidence suggesting subjects began to experience difficulties with the deeper inheritance hierarchy.

2 Experimental justification

In the structured interview phase of the multi-method approach there was consensus amongst object-oriented developers that inheritance depth affects a programmer's ability to understand object-oriented software [DWB+95]. In the questionnaire phase, the majority of object-oriented practitioners (55% of the 273 responses) agreed that inheritance depth is a factor when attempting to understand object-oriented software [DMB+95]. Of these, the largest proportion (57%) indicated that between four and six levels of inheritance depth is where difficulties begin. Since it is well documented that program understanding is a major factor in providing effective software maintenance and that software maintenance accounts for a large part of the total software development budget, this is a finding that could be of major importance. To investigate the phenomenon in a controlled manner, a series of subject-based laboratory experiments, including a replication, were conducted in an attempt to evaluate the effect of inheritance depth on the maintainability of object-oriented software.

Students and recent graduates were used as subjects. The use of student subjects has been justified by Brooks [Bro80] and adopted by researchers in previous empirical studies, e.g., [LHKS92], [PVB95]. Drawing generalisations from their performance, however, is something that should be carefully considered. For example, Curtis has voiced concern about the use of novice programmers as subjects [Cur86]. On the other hand, the series of experiments was conducted within a multi-method programme of research and it was hoped their results would confirm the findings of the structured interview and questionnaire phases. If confirmatory power was achieved, any conclusions drawn would be more reliable and generalisable. Subsequent studies

should still seek to scale up the findings to the maintenance of larger software systems with professional programmers.

3 Experimental design

The experiments sought to determine if inheritance depth has an effect on the maintainability of object-oriented software. Throughout this article the following definitions apply:

Inheritance depth: the level of a class in the hierarchy where the base class is level 1. Consequently, any class is at level n if it has $n-1$ superclasses. The level of the deepest leaf class is quoted as the depth of the hierarchy.

Maintenance: modification of a software product after delivery to correct faults, to improve performance or other attributes, or adapt the product to a changed environment [Sch87].

Maintainability: the ease with which a software system can be corrected when errors or deficiencies occur, and can be expanded or contracted to satisfy new requirements [Sch87].

Maintainability can be measured in a number of ways, e.g., by using complexity metrics or even subjective evaluations by experts. For this study, in operational terms, a difference in maintainability is to be measured by differences in the times it takes subjects to perform maintenance tasks. If a software system is less or more easy to understand and modify than another, then this difference is expected to manifest itself as differences in performance times. In a controlled experiment on the impact of software structure on maintainability, Rombach [Rom87] reports correlations between complexity measures and staff-hour effort. The times reported here reflect a snap-shot view of maintainability, and relative maintainability could change as the maintenance process evolves.

Regardless of program versions in the first experiment and its internal replication, the maintenance task described to subjects was the same. Regarding program versions, effort was made to ensure the maintenance tasks were sufficiently similar to allow meaningful comparisons. (Table 3 suggests there were no task effects arising from the two programs.) In the second experiment, regardless of program versions, the maintenance task described to subjects was the same.

The programs used are regarded as good representatives of the solution spaces, i.e., they are not contrived and are assumed to resemble the solutions most programmers would adopt.

3.1 Design of first experiment

Standard significance testing was adopted and for the first experiment the stated null hypothesis was:

H_{0-exp1} — The use of a hierarchy of 3 levels of inheritance depth *does not* affect the maintainability of object-oriented programs,

to be rejected in favour of the alternative hypothesis

H_{1-exp1} — The use of a hierarchy of 3 levels of inheritance depth *does* affect the maintainability of object-oriented programs.

Note that there is no direction specified in the alternative hypothesis — it was not predicted whether the effect on maintainability would be positive or negative because: (i) of the varying opinions expressed in the maintenance literature about object-oriented software and (ii) although the depth being empirically investigated borders the range indicated most frequently by practitioners in the questionnaire phase as where difficulties begin to occur, it is not completely within that range. A depth of three was chosen to provide an intermediate reference point between the flat code and the second experiment which provides a depth within the range most frequently indicated.

To test the hypothesis a within subjects randomised block design was used — subjects were matched on object-oriented knowledge and were then randomly allocated into one of two groups, A or B. Group A performed a maintenance task on a program with an inheritance hierarchy while group B performed the identical task on an equivalent version of the program without an inheritance hierarchy (referred to from now on as the 'flat' version). To counter-balance this, the reverse was then carried out: group B performed a similar maintenance task on a second, similar program with an inheritance hierarchy while group A maintained the equivalent flat version (Section 3.1.2 explains this design in full). Counter-balancing the groups in this manner should have eliminated any task direction bias and subsequently any ability effect, but there is always the possibility that counter-balancing can introduce a learning effect. The data is examined for this effect (see Section 4.4).

This traditional experimental design provided a single independent and a single dependent variable. The program version (inheritance or flat) being maintained was the independent variable and the dependent variable was the time taken to complete the maintenance task; the most frequent measure of programmers' efforts on software maintenance is the time taken [Fos91]. Data gathering (discussed in Section 3.1.5) was not limited to this dependent variable to allow an inductive analysis to be performed (discussed in Section 4.4).

With the anticipation of around 30 subjects completing in each group, t-test power curves [Lip90] for a two-tailed significance level of 0.05 indicated that the experimental design had a 0.5 probability of detecting a medium-sized effect (where the difference in means is 0.5 of a standard deviation) and a 0.86 probability of detecting a large sized effect (where the difference in means

is 0.8 of a standard deviation). With pairing, the anticipation was that these power levels would be improved upon.

3.1.1 Procedure

The first experiment was performed through a taught postgraduate conversion course in information technology. All of the students (see Section 3.1.2) enrolled in an object-oriented programming class using C++ which was intensively taught over a four week period with approximately nine hours of supervised practical time every week for the first three weeks and five hours in the last week. Students were taught the concepts of object encapsulation, inheritance, message passing, and polymorphism, a working knowledge of which was required to complete the maintenance tasks. Practical exercises were based on these concepts, with students designing and implementing their own classes and inheritance relations and integrating these with existing code.

Students consented to their practical work being used for research purposes and the practical tests/experiments, constituting 60% of the final class mark, were conducted during the final week of the class. For each practical test, every student was given a sheet detailing the experimental instructions, a packet containing the maintenance task, and a second packet containing a listing of the source code. The experimental instructions were also explained verbally at the beginning. (The only other information given was that different versions of the program existed, stated to reduce students concern about their relative performance during an individual test.)

The procedure followed for each of the two practical tests was:

1. Subjects were allowed five minutes to read the instructions and ask questions. When this time had passed and all subjects indicated they were happy with the instructions, they were instructed to open packet 1.

2. Packet 1 contained the maintenance task the subjects were to attempt. Subjects were given a further ten minutes to read the task and ask questions. Again, when this time had passed and all subjects had indicated they were happy with the maintenance task, they were instructed to open packet 2.

3. Packet 2 contained the experimental code listing. Once packet 2 was opened, data recording began and each subject had up to 1 hour 45 minutes to complete the maintenance task and compile and execute the code until the program output matched the required output provided. When subjects were of the opinion that they had completed the task a monitor checked their work. If the output was correct, data recording was terminated; if not, the subject was asked to continue with the modification.

Group	Experiment 1a	Experiment 1b
A	Program 1 inheritance version	Equivalent flat version of program 2
B	Equivalent flat version of program 1	Program 2 inheritance version

Table 1: Group allocations to tasks in the first experiment

After completing the maintenance task, subjects were asked to complete a debriefing questionnaire before leaving. The questionnaire elicited personal details, programming experience, and impressions of the maintenance task just attempted, e.g., the overall task difficulty, what approach to the modification was taken, and what aspect caused the most difficulty.

3.1.2 Subjects

Thirty one students enrolled in the object-oriented programming course, all of whom had completed a ten week class in imperative programming using Turbo Pascal. Each subject sat two multiple choice tests (counting for the other 40% of the class mark) which assessed their object-oriented programming knowledge gained from the class. The subjects were distributed into two groups (16 subjects in group A and 15 subjects in group B) by matching pairs of subjects on the results of these two multiple choice tests and then randomly assigning one to each group: this pre-screening matching was performed to reduce subject variability across the groups.

The two groups were counter-balanced across the program versions with or without inheritance as illustrated in Table 3.1.2. Allocation in this manner ensured that all subjects performed a maintenance task to both a flat program version and an inheritance program version. Subjects who did not complete the task could not be included in the statistical analysis because the nature of the study prevented subjects from continuing after the allocated time period. The efforts made by these subjects, however, have been taken into account.

3.1.3 Maintenance tasks

There were two programs to be modified; each was designed in an object-oriented fashion and then implemented in C++. Both programs were simple database systems which allowed records to be created, displayed, modified, and deleted. The first system stored information on two types of university staff and students via the classes Lecturer, Secretary, and Student. Figure 1 displays the inheritance hierarchy for this database system. The classes Staff and Student inherit from the Univ_Community class, Lecturer and Secretary inherit from Staff, and Professor (to be added) inherits from Lecturer. The classes Univ_Community and Staff are abstract classes: there are no instances of these classes, they merely have the abstract features common to the

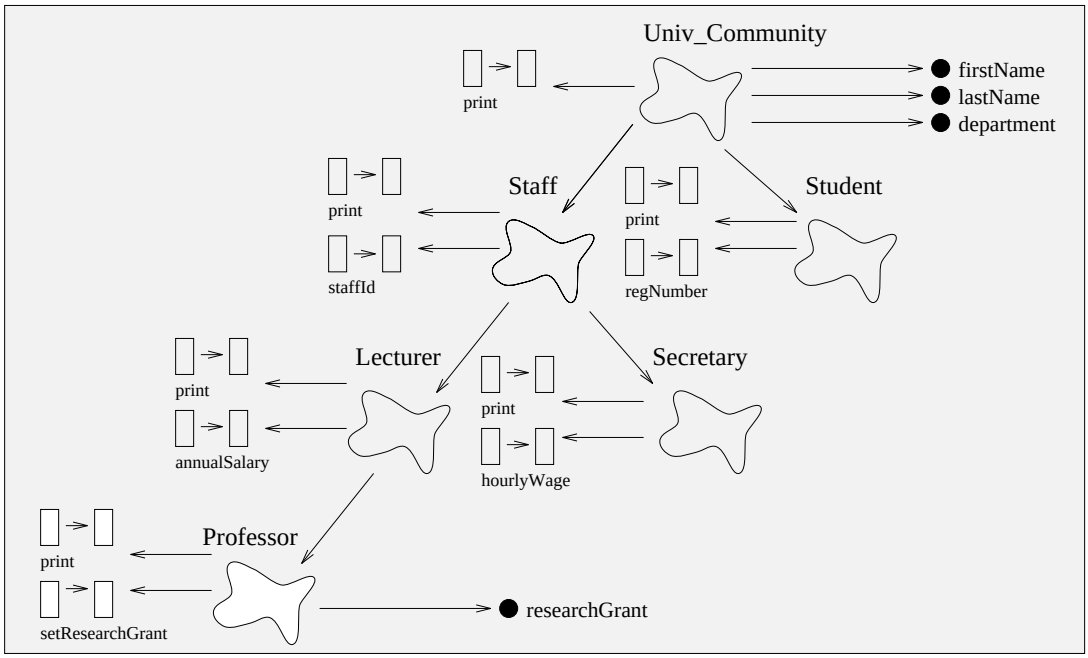

Figure 1: Inheritance hierarchy of database system for university staff and students.

specialisation classes. Instances of Lecturer, Secretary and Professor can receive the message staffId, and the member function in the super-class Staff will be executed. Member functions in any of the subclasses can manipulate the instance variables firstName, lastName, and department by means of the appropriate member functions in the superclass. Finally, each class overloads the member function print to implement its own version. The second system stored information on three types of written work via classes Book, Conference, and Thesis. The inheritance hierarchy for this system was similar to that of the university database, as were the number of fields per class.

Two versions of each system were used, a flat and an inheritance program version. The equivalent flat program versions were created by removing all the inheritance links between the classes in the hierarchy and adding the data members and individual member functions to each class which had previously inherited them. Any abstract classes were then deleted, leaving a 'flattened' but equivalent version of the inheritance hierarchy. The flat program versions were each about 390 lines of code (simple line count, including approximately 25 comment lines — used to identify C++ constructs not class relationships, e.g., 'lecturer constructor', 'assign initial values'). The inheritance program versions were each about 360 lines of code (approximately 35 comment lines). The inheritance depth for each system was three.

To test the hypothesis about the maintainability of object-oriented software, maintenance

tasks were devised which introduced new requirements (in this case, increasing the amount of information the database could store). The subjects' task was to add a single class to their system. A Professor class had to be added to the university system, and a Phd_Thesis class to the library system. The Professor class was to consist of seven fields, some of which are shown in Figure 1, and was intended to be specialised from class Lecturer. The Phd_Thesis class was also to consist of seven different fields and was intended to be specialised from class Thesis. These two tasks were designed to be similar. In line with common programming practices each class was expected to have: (i) its member variables declared as private, (ii) a constructor, (iii) a destructor, and (iv) public member functions (although the required output could be obtained without all of these practices being adhered to). Subjects had then to create an instance of their new class with initial and default values, modify some of these values, and then display the object. Regardless of the program version (inheritance or flat) the maintenance task was the same.

3.1.4 Materials

Each subject was given the following experimental materials:

- a workstation (such that subjects in the same group were not sitting next to each other),
- full experimental and maintenance task instructions,
- complete source code listing of the program (paper-based and on-line), and
- test-data to determine successful task completion.

The environment used was Sun-Sparc workstations, Sun C++ compiler, and the GNU Emacs editor. The experiment was run under laboratory conditions: there was no form of communication between the subjects. They were, however, allowed access to their class textbook [Ski92].

3.1.5 Data collection

The data was automatically collected by a highly controlled environment designed specifically for this study. Each subject was required to start a shell script which provided a workstation prompt with their login name and the time. This script was kept running throughout the experiment and it recorded the process the subject adopted towards the modification; this allowed the reader of the typescript to decipher, for example, how long was spent on a particular problem.

Another shell script was introduced which, while compiling the subject's files to generate the executable, automatically copied each file with a time stamp to a backup directory. This meant

the number of compilations could be calculated and also allowed examination of each subject's solution as it was written and compiled from one stage to the next.

In summary, the data collected from conducting each experiment for any given subject was: (i) the time to complete the task, (ii) automatic file backups, (iii) a script of the subject's experimental procedure, (iv) the final version of the subject's solution, and (v) answers to the debriefing questionnaire.

3.1.6 Pilot study

A pilot study, using four academic staff, was conducted to: (i) find introduced assumptions in the experimental materials, (ii) find mistakes in the experimental procedure, (iii) test that the experimental instructions were clear, (iv) check that the tasks were of reasonable complexity, but that they could be completed well within the allotted time, (v) ensure performance of the automated data collection techniques, and (vi) attempt to identify any other unforeseen circumstances.

No significant issues were encountered during the pilot study, but subjects did require clarification on several points in the instructions, e.g., two subjects mentioned that the description of the required program output was not specific enough. The instructions were subsequently amended to make them clearer.

3.2 Design of internal replication

An internal replication was conducted to confirm the direction of the findings of the first experiment. It was decided to perform the replication with more experienced programmers — it was planned and executed relatively soon after the first experiment and before its results were known. With the anticipation of around 15 subjects completing in each group, t-test power curves [Lip90] for a one-tailed significance level of 0.05 indicated that the experimental design had a 0.4 probability of detecting a medium-sized effect and a 0.7 probability of detecting a large-sized effect. Twenty nine subjects, a mixture of BSc. Computer Science students going into final (fourth) year and new graduates, volunteered to participate. All subjects were expected to be well versed in C programming. The subjects participated in a week long intensive C++ course, during which the internal replication using the library database system was performed. One half of the subjects maintained the flat version; the other half the inheritance version. No internal replication was performed using the university database system because the subjects were to participate in the second experiment which involved using a deeper inheritance hierarchy (see Section 3.3).

Subjects were randomly allocated to two groups in the same manner as that detailed in

Group	Internal Replication	Second Experiment
A	Inheritance version	Equivalent flat version of 5 level hierarchy
B	Equivalent flat version	Inheritance version with 5 levels

Table 2: Group allocations to tasks for the replication and second experiment

Section 3.1.2, but were blocked across their average Computer Science exam marks. The groups were counter-balanced across program versions with or without inheritance as illustrated in Table 3.2. Allocation in this manner again ensured that all subjects performed a maintenance task to both a flat program version and an inheritance program version, i.e., those that performed with the flat program version in the replication were given the inheritance program version in the second experiment and vice versa. The procedures, materials, and environment were the same as they were for the first experiment (see Section 3.1).

For the internal replication the null hypothesis was stated:

> H_{0-rep} — The use of a hierarchy of 3 levels of inheritance depth *does not* affect the maintainability of object-oriented programs,

to be rejected in favour of the alternative hypothesis

> H_{1-rep} — The results of the internal replication will be in the same direction as the first experiment.

The direction specified in the hypothesis indicates the results of the replication were expected to be similar to the results of the first experiment.

3.3 Design of second experiment

Thirty one subjects participated in the second experiment which tested the effect of a deeper inheritance hierarchy on the maintainability of object-oriented software. (These were the same subjects from the replication plus two students who missed it due to prior commitments.) The procedures, materials, and environment used for the first experiment were kept the same. With the anticipation of around 15 subjects completing in each group, statistical power estimates were similar to that for the internal replication.

The system used for this experiment was a larger version of the university database system from the first experiment (see Figure 1). The inheritance hierarchy was extended to include more members of the university community: undergraduate student, postgraduate student, technician, senior technician, and supervisor classes were incorporated into the software. In

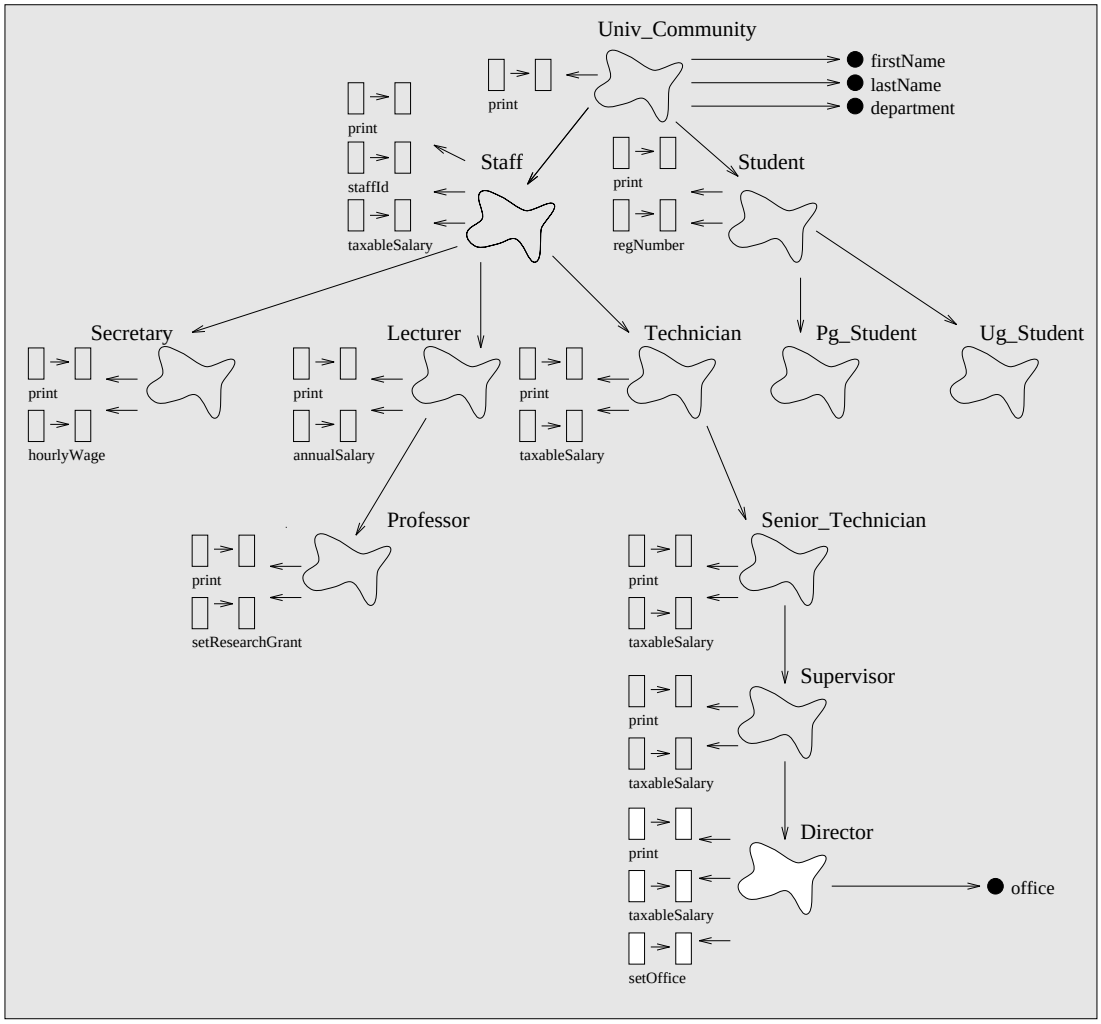

Figure 2: Hierarchy with 5 levels of inheritance for second experiment.

addition, member functions were introduced so that wages and salaries could be calculated for the university employees. Figure 2 displays the inheritance hierarchy for this system. Again, two versions of the system were constructed: a flat program version and an inheritance program version. The inheritance depth for this system was 5. The inheritance program version was approximately 800 lines of code (approximately 90 comment lines), distributed in 12 classes (again, each class was distributed in a header and implementation file) and a main file. The flat program version, constructed in the same manner detailed in Section 3.1.3, had 3 fewer classes (the abstract classes which were deleted), but was around 300 lines longer (approximately 80 comment lines).

The maintenance task for this more complex system was again devised to meet new requirements. The task involved adding a new class, Director, which was expected to be specialised

from class Supervisor (as detailed in Figure 2). Once more the task required member functions to create, modify, display, and delete instances of the class. In addition, a member function had to be written to calculate the taxable salary for Director. Each subject then had to create an instance of their new class and send it messages to invoke actions to meet the required program output. For the second experiment the null hypothesis was stated:

> H_{0-exp2} — The use of a hierarchy of 5 levels of inheritance depth *does not* affect the maintainability of object-oriented programs,

to be rejected in favour of the alternative hypothesis

> H_{1-exp2} — The use of a hierarchy of 5 levels of inheritance depth *does* affect the maintainability of object-oriented programs - subjects maintaining the inheritance program version will take longer than those subjects maintaining the flat program version.

For this hypothesis a direction was provided because the depth being empirically investigated is within the range indicated most frequently by practitioners where difficulties begin to occur [DMB+95].

4 Experimental results

This section details subjects' mean completion times for the maintenance tasks and provides an interpretation of the discovered timing trends.

4.1 First experiment

The timing data collected for the first experiment is presented in summarised form in Table 3. Column two gives the mean time ($\overline{X}_{time}$), column three gives the standard deviation (S_{time}), columns four and five give the minimum and maximum times, column six gives the number of observed times (N), and column seven gives the number of incomplete times (Inc.). Rows one and two present the summary for the first run using the university database system. Rows three and four present the summary for the second run using the library database system. Note that the average times for the inheritance and flat program versions are very similar for the two software systems which suggests that there were no task effects. (The difference in average times for flat versus inheritance for each software system are in the same direction, although they are not statistically significant). In addition, rows five and six present the grouped mean flat and inheritance times for the two runs. Examination of these times shows a mean difference of 11.4 minutes between the total inheritance and flat times.

	$\overline{X}_{time}$	S_{time}	Min.	Max.	N	Inc.
Program 1 Flat	53.1	23.1	26	98	10	5
Program 1 Inheritance	44.1	20.6	18	92	14	2
Program 2 Flat	56.8	23.9	31	100	13	3
Program 2 Inheritance	43.5	20.4	25	102	13	2
Grouped Flat	55.2	23.1	26	100	23	8
Grouped Inheritance	43.8	20.1	18	102	27	4

Table 3: Statistical summary of the first experiment times (minutes)

Statistical tests were then applied. Formal skewness and kurtosis tests were performed and found several of the data distributions to be non-normal at the 95% confidence interval (confidence intervals are provided in [BCM94]). Consequently, to be conservative, non-parametric statistical tests were applied (although for each non-parametric test a similar result was obtained by an alternative parametric tests). A Wilcoxon signed ranks (related) test which takes account of the difference (positive or negative) between paired values, i.e., the performance difference between a subject's time to complete the inheritance program version and the flat program version, was calculated. The statistical test, based upon the 20 subjects who completed both the flat and inheritance program versions, produced a significant result with $p = 0.05$ (two-tailed, $W = 46.5, N = 19, z = -1.95$): 13 subjects performed better on the inheritance than the flat, 6 did the opposite, and 1 achieved the same time for both versions (and so was discounted in the test calculation). Thus we reject the null hypothesis H_{0-exp1} in favour of the alternative hypothesis H_{1-exp1}.

It was of some concern that 11 subjects failed to complete at least one of the tasks within the allotted time. It is important to note the significant statistical result was based on paired observations and thus did not include any of these subjects. Of these 11 subjects, one failed to complete both program versions, 7 subjects completed an inheritance but not a flat version, and 3 completed a flat but not an inheritance program version. On studying the questionnaires and subjects' source code it was found that the most common reason for incompletion was that 6 of the those working with a flat version attempted to develop a solution using inheritance. Most students who successfully completed the task when working with the flat version appeared to use an existing class as a template. Those who attempted to introduce inheritance into the flat version had no such template within their existing code.

Combining those who did complete with those who failed to complete within time provides a performance ratio of 20:9, i.e., approximately 2 out of 3 subjects performed better when maintaining object-oriented software with inheritance.

	$\overline{X}_{time}$	S_{time}	Min.	Max.	N	Inc.
Replication Flat	46.1	20.0	22	97	14	0
Replication Inheritance	35.2	17.0	14	77	13	2

Table 4: Statistical summary of the replication times (minutes)

4.2 Internal replication

The timing data collected is summarised in Table 4 in the same format used in Table 3. The results were in the same direction as the first experiment with a difference in time (10.9 minutes) between the two groups; this is similar to that of the first experiment. Note that the average times for the replication groups are faster than the first experiment by 9.1 minutes for the flat program version and 8.6 minutes for the inheritance program version. This improvement in performance is interpreted as being due to the replication subjects' greater programming experience.

A Wilcoxon rank sum (unrelated) test was calculated for these times because, unlike the first experiment, there was no paired value available for comparison. The statistical result was $p = 0.07$ (one-tailed, $W = 151.0, n1 = 14, n2 = 13, z = -1.51$), but is arguably close enough to the 0.05 level to provide confirmatory power for the first experiment result. We reject the null hypothesis H_{0-rep} in favour of the alternative hypothesis H_{1-rep}.

In this experiment there were 2 incompletions. Questionnaire data suggests that these 2 subjects suffered significant problems with C++ inheritance syntax.

4.3 Second experiment

Table 5 presents the collected timing data in the usual summarised form. Note that the direction of the mean times matches the predicted direction of the hypothesis. Cross checking the mean times from the replication and this experiment (see Tables 4 and 5) shows that while the mean time for the flat group has increased only marginally (approximately 3.5 minutes), the mean time for the inheritance group has increased substantially (on average approximately 19.8 minutes longer per subject). Possible reasons for this large turn around are discussed below.

A Wilcoxon rank sum test (unrelated), however, did not show significance between these mean times $p = 0.27$ (one tailed, $W = 217.5, n1 = 15, n2 = 15, z = -0.62$), and hence the

	$\overline{X}_{time}$	S_{time}	Min.	Max.	N	Inc.
Flat	49.6	21.3	15	92	15	1
Deeper inheritance	55.0	23.9	33	115	15	0

Table 5: Statistical summary of the second experiment times (minutes)

null hypothesis, H_{0-exp2}, cannot be rejected. When the null hypothesis is not rejected, it is important to consider power levels. The design had a good chance, 0.7, of detecting a large effect but only 0.4 of a probability of detecting a medium-sized effect. These power estimates are further weakened by the need that arose to use non-parametric statistics. So a small to medium-sized effect may well exist: the second experiment simply did not have the necessary statistical power to draw conclusions one way or the other regarding the existence of a small to medium-sized effect. On the other hand, the direction of the mean times has reversed for this experiment. This is an important finding and worth exploring.

There was one incompletion in this experiment. The collected source code shows that this subject attempted to reconstruct a complete inheritance hierarchy from his flat code, so this incompletion does not bias the results.

4.4 Inductive analysis and interpretation

Figure 3 displays the spread of the collected times through the use of boxplots (see [CCKT83] for a full description). The first two boxplots represent the times of the first experiment for the inheritance group ($N = 27$) and the flat group ($N = 23$); boxplots three and four represent the inheritance group ($N = 13$) and the flat group ($N = 14$) times for the replication; finally, boxplots five and six represent the inheritance group ($N = 15$) and the flat group ($N = 15$) times for the second experiment. The boxplots show similarity between the spread of the data in the first experiment and the replication and also show the difference in performance between flat and inheritance groups as the depth of the hierarchy is increased. Figure 4 demonstrates more clearly trends in performance.

Of immediate interest is that subjects' relative performance deteriorated on the inheritance program version with a deeper hierarchy.

So what possible interpretations can be placed on the data other than the hypothesis-related interpretation that working with a deeper hierarchy does affect the maintainability of object-oriented programs.

We discount program size as a predictor of performance. Despite a near trebling in size of the flat program versions between the internal replication and the second experiment, flat times were not worse on average on the second experiment. Subjects were not required to understand or search through every line of code; rather they were required to decide which class to specialise from or use as a copy template and they needed only to understand the code related to this selection.

In experiment 1 and its internal replication, it has to be acknowledged that little thought was required to decide which class to specialise from (the superclass) or to use as a copy template. In

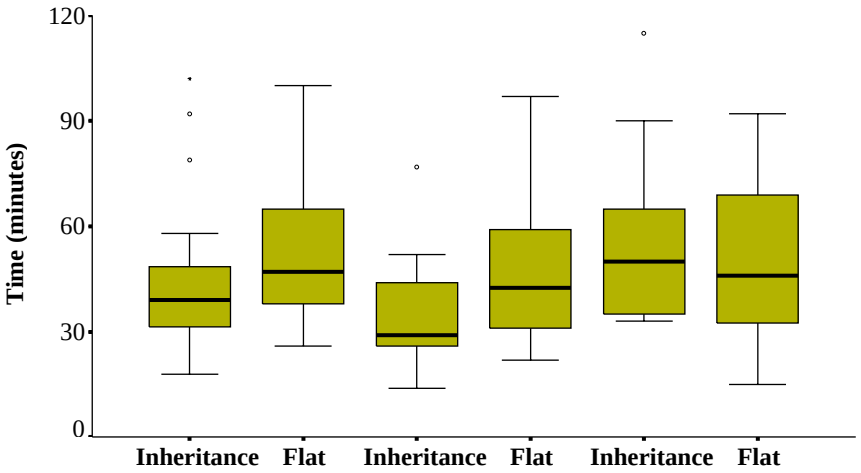

Figure 3: Boxplots of completion times for the first experiment (left), the internal replication (centre), and the second experiment using a deeper inheritance hierarchy (right)

the second experiment, however, the flat program version had 9 classes to consider, a doubling from the first experiment, and the inheritance program version had 12 classes to consider, again a doubling from the first experiment. So subjects were initially faced with a search problem which can have been solved in either a satisfying or optimising way. This search problem was exacerbated by the fact that subjects were not provided with a conceptual model of the domain nor supplied with a strategy on which to base a selection of superclass or copy template.

So a possible interpretation is that experiment 1 and its internal replication simply revealed the modifiability advantage of inheritance which was then cancelled out in the second experiment by the more demanding search problem generated through a greater number of classes interconnected through the inheritance mechanisms. So it need not be depth per se that would cause a performance deterioration: a shallow but broad inheritance hierarchy could just as easily result in a demanding search problem.

Subjects' questionnaire responses provide evidence relating to this interpretation. Table 6 indicates the number of comments or indications received under several categories for the flat and inheritance program groups. These categories are:

1. Problems choosing superclass or class to use as copy template
2. Problems tracing in the inheritance hierarchy
3. Problems with virtual functions
4. Problems with lack of provided conceptual model

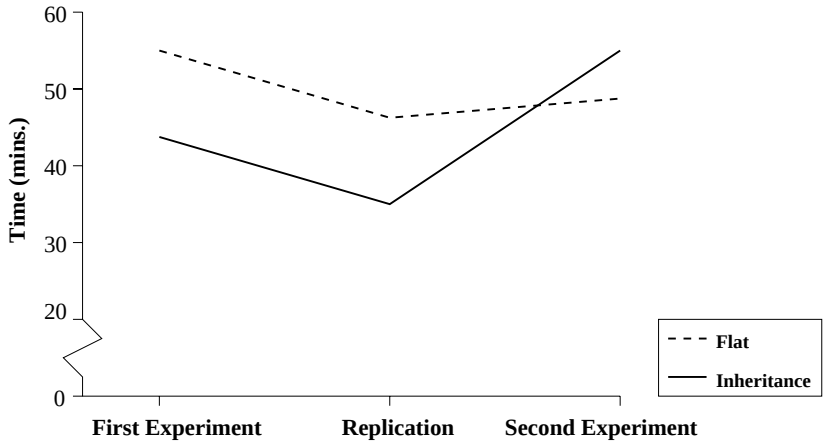

Figure 4: Average completion times for the first experiment, the internal replication, and the second experiment using a deeper inheritance hierarchy

5. Adopted an optimising selection strategy of 'most in common'
6. Despite indicating 'most in common', chose a less than optimal class
7. Adopted a satisfying strategy of 'a good basis'
8. Derivation class selection time less than 5 minutes
9. Derivation class selection time between 5-10 minutes

This approach, of drawing up tables of data in order to detect common patterns of behaviour with a view to understanding underlying processes, is called by us an inductive analysis approach (though it may be called a qualitative analysis by others). The data in the tables may be drawn from questionnaire data or any other measurement taken: within this paradigm it is important to give due attention to the sorts of additional data that should be gathered over and above the data to be used in formal statistical testing. Patterns may be detected directly from any table drawn up or data-mining tools may be applied to the table. The emphasis of the paradigm is on explaining what took place: this provides a check on operational definitions explicitly or implicitly bound up in any null hypotheses.

Note that not every subject responded to every question on the questionnaires and that some subjects are recorded more than once in Table 6 (but not more than once per category). In the first 4 categories of the table, 2 inheritance subjects are recorded under (1) and (4). Also note that our estimate of inter-rater reliability would suggest that an odd comment could be classified under a different category, e.g., a comment about virtual functions could be categorised

as a problem with tracing and a problem with tracing could be categorised as a problem with choosing superclass.

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
flat	1	0	1	1	6	0	1	4	1
inheritance	4	5	4	6	7	3	1	1	7

Table 6: Frequency of comments and other indicators from questionnaires

Assuming the responses are representative, Table 6 suggests that most subjects adopted an optimising selection strategy but that it took the inheritance program group typically at least another 5 minutes to make the selection. Moreover, 3 subjects from the inheritance program group settled on the wrong class despite claiming to have used an optimising strategy and 5 subjects from the inheritance program group essentially invented conceptual models of the domain that meant they chose not to use the superclass with most in common. In contrast to the first experiment and its internal replication where there was considerable (but not complete) agreement on choice of superclass, in the second experiment, inheritance program group subjects can be categorised into three groups: subjects inheriting from class Lecturer (2 subjects), subjects inheriting from class Staff (9 subjects), and subjects inheriting from the 'most in common' class Supervisor (4 subjects). So there is a reasonably clear indication of how more demanding the search problem was for the inheritance program group. We believe our data also supports Dvorak's ideas on conceptual entropy [Dvo94] : all systems that are frequently changed characteristically tend towards disorder, a term recognised as entropy. In object-oriented systems

> "conceptual entropy is manifested by increasing conceptual inconsistency as we travel down the hierarchy. That is, the deeper the level of the hierarchy, the greater the probability that a subclass will not consistently extend and/or specialise the concept of its superclass." [Dvo94].

Dvorak identified this concept through an experiment where subjects were to construct a class hierarchy from class specifications: the deeper the hierarchy got the less agreement there was between subjects about a class's placement in the hierarchy. Essentially, a similar effect has been found here.

But this search problem alone is not enough to explain away all the worsening in performances for the inheritance program group.

Table 6 indicates that most subjects in the inheritance program group reported that they suffered from problems due to tracing or virtual functions. (The data for categories (2) and (3) are for separate subjects.) These problems probably go some way toward explaining the

rest of the worsening in performances for the inheritance program group. As noted earlier, one difficulty that affects program understanding, and hence maintenance, is the presence of delocalised plans, where pieces of code that are conceptually related are physically located in non-contiguous parts of the program. According to Wilde *et al.*, the mechanism of inheritance creates further opportunities for delocalisation [WMH93]. One such related difficulty is that understanding a single line of code may require tracing a line of method invocations through an inheritance hierarchy. In a shallow hierarchy this may not represent a large overhead, but as the hierarchy becomes deeper the overhead is likely to increase. In the case of a maintainer who wants to view the actual implementation of a method, tracing the line of invocations to its source must be conducted. Such tracing may have affected some subjects' maintenance times.

Note that no subject from the first experiment or its internal replication commented on problems choosing superclass/class to use as a copy template or problems tracing through the hierarchy.

The subject who took by far the longest time on the inheritance program group made several tries at different points in the hierarchy. If this subject's datum is excluded the average time for the inheritance program group would almost match the average time for the flat program group. We have no reason to exclude this datum: the subject's behaviour is a particularly poignant example of conceptual entropy.

To summarise:

1. Program size is discounted as a predictor of performance.

2. The inheritance program group working with a deeper inheritance hierarchy had a more demanding search problem when choosing a superclass but that this alone does not account for the relative deterioration in performances.

3. Conceptual entropy will arise when programmers are forced to create their own conceptual models of the domain or if they are given a free choice between satisfying and optimising strategies when specialising in an inheritance hierarchy.

4. Problems with tracing and virtual functions go some way toward explaining the deteriorations in performances.

5. The more demanding search problem and the problems with tracing and virtual functions, together, probably explain the general relative deterioration in performances.

5 Threats to validity

5.1 Threats to internal validity

A major concern within any empirical study is that an unobserved independent variable is exerting control over the dependent variable(s), a possibility which must be minimised. Three such threats have been identified: (i) selection effects, (ii) maturation effects, and (iii) instrumentation effects.

1. Selection effects are due to natural variations in subject performance (see, e.g., [Bro80]). An example of this is presented in [DBM+94] where the majority of 'high ability' subjects were randomly assigned to one of two groups, something which obviously biased the results of the study. Such bias was catered for in this study by creating subject groups of equal ability (as detailed in Sections 3.1.2 and 3.2).

2. Maturation or learning effects are caused by subjects learning as an experiment proceeds. The threat here was that subjects would learn from the first run and that their performance on the second run would be biased. The data was analysed for this and no significant effect was found.

3. Instrumentation effects may result from differences in the experimental materials employed. In this study such effects were likely to arise from differences in the presented software systems and maintenance tasks. Although an explicit attempt was made to ensure as much similarity as possible, such variation can be difficult to avoid. The collected timing data for the first experiment are very similar across the two runs; the internal replication repeated these results. This increases confidence that any such effect was minimised. Instrumentation effects also appear minimal between the replication and second experiment — the increase of mean time for the inheritance group would have been similar for the flat group otherwise.

So there is no evidence suggesting that these threats to internal validity have impacted on the results of the study.

5.2 Threats to external validity

The greater the external validity, the more the results of an empirical study can be generalised to actual software engineering practice. Three threats to external validity have been identified which limit the ability to apply any such generalisation: (i) subject representativeness, (ii) the size of the software systems used, and (iii) maintenance is a process, and only the implementation phase of the process has been considered in these experiments.

1. The subjects who participated in the experiments may not be representative of software professionals. Although the participants in the replication and second experiment were a mixture of final year students and new graduate computer scientists and were classed as more experienced programmers, they cannot be categorised as experienced software professionals. For pragmatic considerations, having students as subjects was the only viable option for the laboratory-based experiments.

2. The software systems used for the experiments were not large and may not be representative of real software systems. The inheritance depth used in these software systems is representative of real inheritance hierarchies, however — see the characteristics of object-oriented class hierarchies presented in [CK94]. Furthermore, it may be that to control and isolate the effect of inheritance on the maintainability of object-oriented software, small systems are required otherwise the effect may become too difficult to detect. As noted by Tiller, more control exerted over an experiment is gained only at the expense of its realism [Til91] — an attempt to achieve as fine a balance as possible was made.

3. Although maintainability of software is best evaluated with respect to the entire maintenance process, laboratory-based experimentation on such a scale is not practical; this study has concentrated on the implementation phase of the maintenance process.

The first and second threats to external validity are common to many reported empirical studies, e.g., [HHL90], [LHKS92], [PVB95]. It is argued there is justification to conclude that because the effect of inheritance has been consistently reported across the multi-method programme of research [Dal96], there is less of a threat to external validity. To fully overcome these threats a replication package is proposed as further work in order to allow other researchers to conduct external replications using different subjects, variables, and procedures (see [Cha88] for a detailed discussion about making generalisations).

6 Conclusions

This empirical study should be of interest to those designing and maintaining object-oriented software. The results suggest that when it is obvious which class should be used to specialise from and when little tracing up through hierarchies is demanded, then inheritance provides gains in modifiability, i.e., object-oriented software is more maintainable than equivalent object-based software. On the contrary, when conceptual entropy exists (when either the conceptual model of the domain has not been not provided or other strategies for specialising have not been specified, e.g., 'most in common') and when tracing up through hierarchies is required for

sound comprehension, then modifiability gains are cancelled out, i.e., object-oriented software is no more maintainable than object-based software. One of our subjects provided a particularly poignant example of conceptual entropy by attempting specialisations at several points in the hierarchy.

An interpretation based solely on our experimental hypotheses would be misleading. Deteriorating performances were not simply down to depth and increased tracing difficulties: conceptual entropy also played a part and one could imagine shallow and broad hierarchies suffering from conceptual entropy as much as narrow and deep. So the inductive analysis was a vital component of our research.

While threats to the external validity have been identified, it is argued that because the results have been confirmed across a multi-method programme of research, these threats are reduced. Subsequent experimentation, however, should make use of larger software systems using professional programmers as subjects. Such experimentation might also consider other categories of maintenance and other aspects of the overall maintenance process. It is not at all obvious that object-oriented software is going to be more maintainable in the long run.

Acknowledgements

The authors wish to acknowledge the efforts of those who participated in the experiments. Thanks are extended to Pete Hendry and Dave Lloyd for their technical assistance.

References

[BCM94] A. Brooks, D. Clarke, and P. McGale. Investigating stellar variability by normality tests. *Vistas in Astronomy*, 38:377–399, 1994.

[Bro80] R. Brooks. Studying programmer behavior experimentally: The problems of proper methodology. *Communications of the ACM*, 23(4):207–213, April 1980.

[Bur95] A. Burgess. Finding an experimental basis for software engineering. *IEEE Software*, 28(3):92–93, 1995.

[CCKT83] J. Chambers, W. Cleveland, B. Kleiner, and P. Tukey. *Graphical methods for data analysis*. Wadsworth International Group, first edition, 1983.

[Cha88] A. Chapanis. Some generalisations about generalisation. *Human Factors*, 30(3):253–267, 1988.

[CK94] S. Chidamber and C. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[Cur86] B. Curtis. By the way, did anyone study any real programmers? In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers: First Workshop*, pages 256–262. Ablex Publishing Corporation, 1986.

[CvM93] R. Crocker and A. von Mayrhauser. Maintenance support needs for object-oriented software. In *Proceedings of the International Computer Software and Applications Conference*, pages 63–69, November 1993.

[Dal96] J. Daly. *Replication and a Multi-Method Approach to Empirical Software Engineering Research*. PhD thesis, Department of Computer Science, University of Strathclyde, Glasgow, 1996.

[DBM+94] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. Verification of results in software maintenance through external replication. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 50–57, September 1994.

[DBM+95] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. A multi-method approach to performing empirical research. *Software Engineering Technical Council (TCSE) Newsletter*, 14(1):SPN10–12, Fall 1995.

[DMB+95] J. Daly, J. Miller, A. Brooks, M. Roper, and M. Wood. Issues on the object-oriented paradigm: A questionnaire survey. Research report EFoCS-8-95, Department of Computer Science, University of Strathclyde, Glasgow, 1995.

[Dvo94] J. Dvorak. Conceptual entropy and its effect on class hierarchies. *IEEE Computer*, 27(6):59–63, June 1994.

[DWB+95] J. Daly, M. Wood, A. Brooks, J. Miller, and M. Roper. Structured interviews on the object-oriented paradigm. Research report EFoCS-7-95, Department of Computer Science, University of Strathclyde, Glasgow, 1995.

[Fos91] J. Foster. Program lifetime: A vital statistic for maintenance. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 98–103, 1991.

[HHL90] S. Henry, M. Humphrey, and J. Lewis. Evaluation of the maintainability of object-oriented software. In *IEEE Conference on Computer and Communication Systems*, pages 404–409, September 1990.

[JKZ94] P. Jüttner, S. Kolb, and P. Zimmerer. Integrating and testing of object-oriented software. In *Proceedings of the European Conference on Software Testing, Analysis, and Review*, pages 13/1–13/14. Siemens AG, 1994.

[Jon94] C. Jones. Gaps in the object-oriented paradigm. *IEEE Computer*, 27(6):90–91, June 1994.

[KGH+94] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change impact identification in object-oriented software maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 202–211, September 1994.

[LHKS92] J. Lewis, S. Henry, D. Kafura, and R. Schulman. On the relationship between the object-oriented paradigm and software reuse: An empirical investigation. *Journal of Object-Oriented Programming*, 5(4):35–41, 1992.

[Lip90] Mark W. Lipsey. *Design Sensitivity, Statistical Power for Experimental Research.* SAGE Publications, 1990.

[LMR92] M. Lejter, S. Meyers, and S. Reiss. Support for maintaining object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1045–1052, December 1992.

[PB94] C. Ponder and B. Bush. Polymorphism considered harmful. *ACM SIGSOFT, Software Engineering Notes*, 19(2):35–37, April 1994.

[PVB95] A. Porter, L. Votta, and V. Basili. Comparing detection methods for software requirements inspections: A replicated experiment. *IEEE Transactions on Software Engineering*, 21(6):563–575, June 1995.

[Rom87] H. D. Rombach. A controlled experiment on the impact of software structure on maintainability. *IEEE Transactions on Software Engineering*, 13(3):344–354, March 1987.

[Sch87] N. Schneidewind. The state of software maintenance. *IEEE Transactions on Software Engineering*, SE-13(3):303–310, 1987.

[Sch95] S. Schneberger. Software maintenance in distributed computer environments: System complexity versus component simplicity. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 304–313, 1995.

[Ski92] M. Skinner. *The C++ primer: a gentle introduction to C++.* Silicon Press and Prentice Hall, first edition, 1992.

[SPL+88] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, 1988.

[Til91] D. Tiller. Experimental design and analysis. In N. Fenton, editor, *Software Metrics — A Rigorous Approach*, pages 63–78. Chapman and Hall, 1991.

[WH92] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, December 1992.

[WMH93] N. Wilde, P. Matthews, and R. Huitt. Maintaining object-oriented software. *IEEE Software*, 10(1):75–80, 1993.