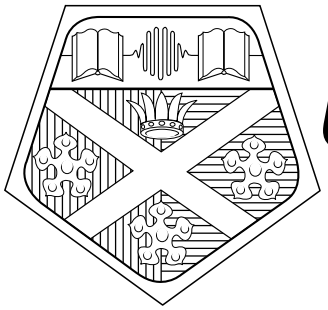
REPLICATION AND A MULTI-METHOD APPROACH TO EMPIRICAL SOFTWARE ENGINEERING RESEARCH

Submitted to the Department of Computer Science,
University of Strathclyde, Glasgow
for
the Degree of Doctor of Philosophy.

By
John William Daly
March 1996



THE UNIVERSITY OF STRATHCLYDE IN GLASGOW

Abstract

Empirical research is vital if software engineering technology is to be meaningfully evaluated — without it only guesses can be made at the competing merits of different approaches. Unfortunately, as this thesis demonstrates, much of existing empirical work contains some kind of weakness because the approaches used to conduct the research are inadequate in many ways. To combat this problem, a more methodical approach to empirical software engineering research is required. This thesis examines replication and a multi-method approach which, it is argued, can improve the quality of results achieved by performing empirical research.

External replication, where independent researchers seek to check and improve the findings of an experiment, is discussed. The results of an external replication which did not repeat the results of the original study are presented; they demonstrate the importance of replicating experiments to gain confidence in the findings. As a result, a set of general recommendations are made for researchers conducting external replications, and conducting and reporting subject-based experiments.

A multi-method approach to empirical research is proposed which comprises of a series of empirical studies being evolved from an initial exploratory study through to laboratory-based studies involving internal replications. The multi-method approach allows attention to be paid to important matters and the results from each study in the series can be confirmatory. Results are presented of an application of the multi-method approach to the object-oriented paradigm. A set of general recommendations are then made for applying the multi-method approach which should enable researchers achieve quality empirical results in which they can have confidence.

It is concluded that if researchers adopt a methodology for empirical software engineering research which integrates the multi-method approach with the technique of replication then, over time, more reliable and generalisable results will be realised.

Acknowledgements

I would like to thank Dr. Andrew Brooks, Dr. James Miller, Dr. Marc Roper, and Dr. Murray Wood for their supervision, encouragement, and stimulating discussions. I believe it is fair to say that without their help and guidance it is likely that this thesis would never have been completed. It has been a pleasure working with you, gentlemen.

There are also several other individuals who have contributed, in one way or another, towards the body of research contained within this thesis. They are: Peter Day, Fiona Ferguson-Smith, Pete Hendry, Dave Lloyd, Fraser Macdonald, Don Millington, Kevin Waite, and Dave Whittington. To you I record my thanks.

I also wish to thank researchers who have been, or still are, PhD candidates within the Computer Science Department as well as many of the departmental staff — our friendly discussions helped me gain confidence in my research and sustain my attempt to complete it.

I must acknowledge the effort my girlfriend has expended putting up with my selfish behaviour during the completion of this thesis; more importantly, for understanding and helping in ways others could not. My deepest thanks, Susie! I also acknowledge the support that my good friends Allan, David, Euan, Jim, Paul, and Stephen have given me during the time it has taken to complete this research. Cheers, lads! And, finally, I wish to thank my mother, father, and family for their eternal faith in my ability to complete this thesis and for helping whenever they could.

The research contained within this thesis has been supported by EPSRC (formerly SERC) under award number 92314532. Due acknowledgement must also be given to the C.K. Marr Trust for additional, and much appreciated, financial support.

List of publications

A number of publications have resulted from the empirical research presented in this thesis:

- [Daly *et al.*, 1994] presents the results of an external replication of a well performed software engineering experiment. The paper was presented by the author at the IEEE International Conference on Software Maintenance in Victoria, Canada.

- [Daly *et al.*, 1995a] details preliminary analysis of the series of experiments testing the effect of inheritance on the maintainability of object-oriented software. The paper was presented by James Miller at the IEEE International Conference on Software Maintenance in Nice, France.

- [Daly *et al.*, 1995b] briefly describes the multi-method approach for performing empirical software engineering research and summarises the results of the three phased programme of research within the object-oriented paradigm.

- [Daly *et al.*, 1995c] presents the findings of a questionnaire survey on object-oriented systems. The paper was presented on behalf of the author by Don Millington at the IEEE Asia-Pacific Software Engineering Conference in Brisbane, Australia.

- [Daly *et al.*, 1996a] reports in full the results of the series of experiments testing the effect of inheritance on the maintainability of object-oriented software. The paper was presented by the author at the 6th Workshop on Empirical Studies of Programmers in Washington D.C., USA. A poster version of the questionnaire survey was also presented [Daly *et al.*, 1996b].

- [Miller *et al.*, 1996] details the strengths and weaknesses of distributing questionnaires for exploratory research by means of electronic newsgroups and compares these to the strengths and weaknesses of other distribution media.

Contents

Chapter 1

Introduction

1.1 Overview

For software engineering, the last quarter of a century has been a time of great change. Appreciating the problems of software was the beginning in the late 1960s and early 1970s. During the 1970s and 1980s, structured design dominated, the guidelines of which, if adhered to, promised greater understanding and improved maintenance of software [Page-Jones, 1988], [Pressman, 1994]. Then new analysis, design, and implementation paradigms emerged promising more reusable, portable, and less complex software systems [Berard, 1993], [Cox, 1986]. Until recently, however, little evaluative research was conducted to support the claims made of emerging software technologies; with the exception of a few researchers, the software engineering community did not really object. As a consequence, software engineering has accumulated a large body of software technology without any supporting quantitative evidence [Card *et al.*, 1986]. This has led software engineers to use such technologies without fully appreciating their benefits and drawbacks relative to other software technologies or when used in a particular environment.

Fortunately, attitudes are now beginning to change: ever more researchers are placing greater emphasis on evaluative research by empirical means to test the claims made of software technology, e.g., [Basili, 1992], [Fenton *et al.*, 1994], [Glass, 1994], [Votta and Porter, 1995]. Researchers are more aware that without empirical research only educated guesses can be made at the competing merits of different approaches, what environments they are likely to perform well in, and what environments they are

unlikely to perform well in. Although this change in attitudes is a move in the right direction, there are many aspects of conducting empirical research that the software engineering community should give more detailed consideration. For example, that the measures taken to capture the effect under study are appropriate, that empirical studies are repeated to gain confidence in their results, and the ability to generalise the results to the population and other environments. To understand software development better and manage it more cost-effectively, therefore, more methodical empirical research must be conducted [Basili and Reiter, 1981].

Conducting empirical research, however, suffers from certain difficulties. For instance, conducting an empirical study is time consuming and requires a large amount of effort to plan, design, and analyse the collected data; one serious mistake can invalidate the results. For laboratory experiments it can be difficult to obtain subjects with appropriate experience; it can be even more difficult to understand subjects' behaviour during a laboratory experiment. For case studies it can be difficult to convince an organisation to allow evaluation of some aspect of their software process or product. Conclusions drawn from an empirical study may only apply to a specific environment and, subsequently, it may be wrong to generalise the results to other environments. Despite such difficulties, simply relying on intuition about software can be misleading [Burgess, 1995]. Further, reliance on only anecdotal evidence can lead to biased conclusions. As a consequence, it is argued that conducting empirical research is the only way to improve the software process and product [Fenton *et al.*, 1994], [Pfleeger, 1994], [Votta and Porter, 1995].

Yet more than fifteen years ago Curtis argued that although the scientific study of software engineering was young, maturation of empirical methods would improve its rate of progress [Curtis, 1980]. Today, for several reasons, it is fair to question how much more mature (if at all) the empirical methods of software engineering are in comparison. First, there is still not enough empirical work being conducted. Second, too much of the empirical research that is conducted falls into the 'one-shot study' category. Third, researchers are still reporting empirical work that is not as valuable or influential as it might be. In short, there is a need for (i) more empirical research to be conducted and (ii) empirical research to be conducted more methodically. This thesis makes two proposals to improve methodology in empirical software engineering — external replication and the multi-method approach.

1.2 The thesis position

1.2.1 Motivation for thesis

Empirical software engineering research is conducted to help evaluate, predict, understand, control, and improve the software development process or product [Basili *et al.*, 1986]. The motivation for the research presented within this thesis, however, is not so much to improve the software process or product, but more to provide software engineering with a research methodology which improves the quality of results achieved from undertaking empirical research. There are several reasons for this motivation:

1. Much of the empirical research that has been conducted to date contains a weakness because of the empirical methodology used — for example, investigation of a weak hypothesis, insufficient data analysis, or a lack of detailed reporting. It is argued that conducting empirical research provides a basis for the advancement of software engineering. As a consequence, every effort must be made to improve the empirical work that is conducted by making it more methodical and making it more reliable and generalisable.

2. Empirical results should be confirmed before they are accepted by the software engineering community, yet this aspect of empirical research is considered by too few researchers. If techniques which provide confirmatory power, i.e., the notion of providing confidence in the results of an empirical study, can be realised and used by software engineering researchers then, over time, more reliable results will emerge. Hence, over time, these results should have more impact on industrial practice.

3. Much of the empirical research presented within the software engineering literature has not included analysis of the data for alternative interpretations from those offered by the applied statistical tests. With subject-based experimentation, it is probable that more than one interpretation can be placed on the data. Many reported conclusions may have been different if researchers had looked for alternative interpretations.

1.2.2 Proposed research methodology

The motivation for this thesis is to improve the empirical research that is conducted within software engineering. To achieve this goal a research methodology integrating two particular techniques is proposed:

Replication: one method of achieving confirmatory power is to replicate an empirical study. Replication takes two forms: internal and external. Internal replications are undertaken by the original experimenters to increase their own confidence in the results; external replications are undertaken by independent researchers who seek to check and improve on the findings of other researchers and are critical for establishing sound results. An external replication of a well performed software engineering experiment will demonstrate the importance of seeking confirmatory power and will illustrate the need for more methodical reporting of empirical research.

Multi-method approach: a second method of achieving confirmatory power is by investigating a particular hypothesis through the use of different empirical techniques, where the data collected from one technique can then be used to complement data collected from another technique. The multi-method approach can also be regarded as evolutionary — the important issues discovered by an initial study are refined and investigated further by the next study, and so forth. Thus in an evolutionary programme of research, the results from each empirical study may turn out to confirm one another. It will be argued that these results are likely to be more reliable and generalisable.

In addition, consideration is given to state-of-the-art software engineering: the object-oriented paradigm has grown increasingly popular during the 1990's, yet it lacks supporting empirical evidence. This situation provides ideal justification for conducting a programme of empirical research within the object-oriented paradigm to evaluate the multi-method approach.

1.2.3 Contribution of thesis

This thesis provides two contributions to empirical software engineering.

1. The first (reported) external replication of any software engineering experiment demonstrates the importance replication holds in an empirical discipline which uses humans as subjects. Moreover, the replication study demonstrates the importance of confirming empirical results before they are given any significant weight by the software engineering community.

2. The multi-method approach is a new technique for performing empirical software engineering research. An application of the approach to an investigation within

the object-oriented paradigm led to a three phased programme of research which demonstrates its strengths in comparison to a single-method approach.

Lessons learned from conducting an external replication and applying the multi-method approach are catalogued. These lessons should provide reference points for researchers preparing their empirical work to allow external replication, preparing to undertake an external replication, or preparing to conduct their own multi-method programme of research.

In addition, detailed analysis of the collected data for the subject-based laboratory experiments has been performed. This was undertaken to explore alternative interpretations of the data. And, finally, several interesting findings resulting from the application of the multi-method approach to the object-oriented paradigm may become hypotheses for future research.

1.3 Outline of thesis

The remainder of this thesis is partitioned into four distinct parts which, in turn, are divided into one or more chapters:

Part I: BACKGROUND LITERATURE

Chapter 2: Empirical Research Within Software Engineering

A review of the empirical literature is presented which discusses (i) established empirical frameworks and methodologies, (ii) empirical issues that should be given more consideration by researchers, and (iii) the variability that exists in researchers' empirical practices.

Part II: REPLICATION

Chapter 3: Confirmatory Power Through Replication

The notion of confirmatory power is developed and the need for replication is introduced and expanded upon. A framework for external replications of other empirical studies is established.

Chapter 4: An External Replication Study

The findings of an external replication of a software engineering experiment are presented. The results are markedly different from those of the original and reasons for this difference are discussed.

Chapter 5: Evaluation

The lessons learned from conducting the external replication are discussed and a list of guidelines for experimental reporting are presented which, if followed, should better enable other researchers to perform external replications.

Part III: A MULTI-METHOD APPROACH

Chapter 6: Introduction

The multi-method approach for performing empirical research is introduced and discussed. Justification is provided for conducting empirical research within the object-oriented paradigm and a plan of the intended programme of research is presented.

Chapter 7: Phase I: Structured Interviews

Structured interviewing is discussed as a technique for performing empirical software engineering research. The findings of interviews conducted with experienced object-oriented developers are then presented.

Chapter 8: Phase II: A Questionnaire Survey

The utility of questionnaires as a technique for performing empirical software engineering research is discussed. The design, implementation, and results of a questionnaire survey, evolved from the findings of the structured interviews, are then presented.

Chapter 9: Phase III: A Series Of Laboratory Experiments

One of the more important findings from the questionnaire survey and structured interviews is investigated further through a series of subject-based laboratory experiments.

Chapter 10: Evaluation

The strengths and weaknesses of the multi-method approach are discussed. The lessons learned from conducting the programme of research are presented for use by other researchers.

Part IV: CONCLUSIONS

Chapter 11: Conclusions And Further Work

The final chapter summarises the results of this thesis and discusses possible further work. It states clearly the contributions of this thesis and ends with the conclusion that researchers should adopt a methodology for empirical

software engineering research which integrates the multi-method approach with the technique of replication.

Appendix A: Glossary

The glossary contains definitions of the empirical terms used throughout this thesis.

Part I

BACKGROUND LITERATURE

Chapter 2

Empirical Research Within Software Engineering

2.1 Introduction

Although conducting empirical software engineering research is becoming an important part of evaluating new software technology, much of existing software technology has been adopted on the basis of expert opinion and anecdotal evidence, not on the basis of empirical or strong theoretical evidence, e.g., see [Basili, 1992], [Fenton *et al.*, 1994], [Votta and Porter, 1995]. While this can be partially blamed on the fact that software engineering is a relatively new field which has grown quickly over a short period of time, empirical evaluation of such technology should be attempted. Evaluation is usually not performed because the need for scientific confirmation is, unfortunately, outweighed by the software engineering community's reliance on intuition. Yet, in an interview with Burgess, Victor Basili states "... in many of our experiments we have shown that our intuition about software is wrong" [Burgess, 1995]. For example, in their experiments evaluating the efficiency of code reading, functional testing, and structural testing, Basili and Selby claimed to have discovered that professional programmers using code reading detected more software faults and had a higher fault detection rate than other methods [Basili and Selby, 1987] — this was a surprising result to many of the programmers that participated in the experiments who felt they had performed better with the testing techniques. While findings about intuition being misleading strengthen the need for more empirical research, it should be noted that performing

empirical software engineering research is not an exact discipline and there is a need for researchers to consider various concerns.

This chapter presents a literature review whose goal is to make researchers more aware of such concerns. The review begins by examining empirical software engineering papers which establish frameworks or methodologies for conducting empirical work and indicates which of these are most relevant to the research conducted within this thesis. Examination of issues that should be, but rarely are, considered when conducting empirical software engineering research follows. To demonstrate the variability that exists in empirical practice, selected empirical studies are introduced and critiqued (including empirical work conducted within the object-oriented field). The review ends by highlighting deficiencies in current practice. Some of the work detailed in this review is relatively old in terms of software engineering literature. This is an indication of the progress (or lack of it) that has been made during this time.

2.2 Empirical frameworks and methodologies

Basili *et al.* provide the most well known empirical framework within software engineering, the experimentation framework paradigm [Basili *et al.*, 1986]. The experimentation framework paradigm represents a refinement of part of the Goal/Question/Metric paradigm (GQM) [Basili and Weiss, 1984] which, in turn, is a mechanism for defining operational goals and producing a set of metrics from them as required by the Quality Improvement Paradigm (QIP) [Basili and Rombach, 1988]. (QIP and GQM are directed towards tailoring the needs of an organisation to understand and improve their software processes and products). The experimentation framework paradigm has four phases each of which correspond to a phase within the experimentation process, i.e., experiment definition, planning, operation, and interpretation. (Table E.2 in Appendix E reproduces the experimentation framework paradigm in full). Each phase is briefly summarised:

1. Definition: the definition phase contains six parts — motivation, object, purpose, perspective, domain, and scope. There can be several motivations, objects, purposes, and perspectives to an experimental study, e.g., the motivation may be to understand, assess, and improve the effect of a particular software technology.

2. Planning: the planning phase of the experimental process contains three parts — design, criteria, and measurement. Design incorporates the different type of methods that can be applied to the experimental design as well as various types

of statistical methods that can be used. The criteria is dependent on the outcome of the first phase of the experimental process, i.e., different motivations, objects, etc, require the examination of different criteria. The effect of the phenomenon under investigation is captured through various data measurements.

3. Operation: the third phase contains three parts — preparation, execution, and analysis. Preparation may include conducting a pilot study to check the data collection methods and the experimental environment are in order. The data is collected during the execution of the experiment. The analysis of the data may include a combination of both qualitative and quantitative methods.

4. Interpretation: the final phase of the experimental process also contains three parts — interpretation context, extrapolation, and impact. The contexts of the interpretation are the applied statistical analyses, the purpose of the study, and the knowledge that already exists in the research area. Sample representativeness is the major factor for extrapolating the results to other environments. The impact of the study is dependent upon the presentation of the results for feedback, replication of the study, and application of its results.

The experimental framework paradigm is intended to structure the experimental process and provide a classification scheme for existing empirical studies so that they can be better understood and evaluated.

Other researchers have also voiced their concern about the lack of structure for the empirical process within software engineering. Curtis reviews the contribution of measurement and experimentation to the state-of-the-art in software engineering and makes several recommendations to improve measurement and experimental evaluation of software techniques and practices [Curtis, 1980]. The most notable of these recommendations is that experimental results are more impressive when they emerge from a programme of empirical research rather than from a one-shot study. According to Curtis, a programme of research offers the advantages of (i) replication of results, (ii) explication of the important factors which rule the process involved, and (iii) evolution of the measurement and experimental methods used.

Moher and Schneider address the lack of methodology in empirical software engineering by formalising the use of controlled group experimentation [Moher and Schneider, 1982]. The authors identify what they feel are the five most important categories which researchers conducting such experimentation must address: experimental design, subject characteristics, examination environment, performance requirements, and

measures. Moher and Schneider define these as follows,

1. Experimental design: refers to the selection of subjects, the selection of variables and treatment levels, and the assignment of treatments to subjects.

2. Subject characteristics: the attributes of a subject prior to any experimentation.

3. Examination environment: includes factors such as training and materials as well as the physical environment to be used.

4. Performance requirements: consists of tasks the subject must perform and any constraints imposed.

5. Measures of the experiment: includes the objective and subjective means by which the subject characteristics, environment, and performance are evaluated.

Moher and Schneider conclude that to achieve meaningful results, experiments must be properly conducted; that means due consideration of these categories.

Chapanis discusses several factors which can limit an experiment's generalisability including unrepresentative subjects, inappropriate selection of dependent variables, and artifacts which attribute to the actual measurement process [Chapanis, 1988]. Chapanis then recommends two principals of design which will allow findings to be extrapolated to a wider range of situations:

1. Design as much heterogeneity into the study as possible.

2. Replicate the study with variance in subjects, experimental variables, or experimental procedures.

Although this paper is from the human factors literature, the concepts discussed within it strongly apply to experimentation within software engineering.

MacDonell claims that the lack of widespread industry acceptance of much of the research into measurement of software complexity is due to the lack of experimental rigor associated with many of the studies [MacDonell, 1991]. MacDonell examines areas where problems have previously arisen when conducting empirical work and provides recommendations regarding more adequate experimental procedures. These areas are: pre-experiment design, operational definitions, experimental method, subjective assessment, data collection, program sizes, program sample sizes, languages revisited, subjects, confounding factors, statistical validity of predictive relationships,

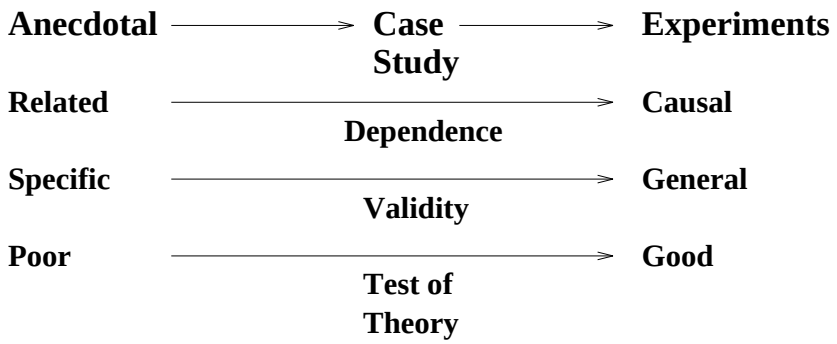


Figure 2.1: Votta and Porter's spectrum of empirical work

result interpretation, and publication of data and assumptions. Although the examples used are mainly concerned with software complexity measurement, MacDonell generalises his recommendations to empirical procedures in most software measurement domains. Hence, this paper could be interpreted as a general review of the problems of empirical software engineering research.

Potts asks why most of the research performed in software engineering has failed to make an impact on industrial practice [Potts, 1993]. The author refers to the research-then-practice approach — a model characterised as "conceive an idea, analyse the idea, advocate the idea" and contrasts this with his industry-as-laboratory approach — identify problems through industrial collaboration and create and evaluate solutions as a research activity. Potts regards the industry-as-laboratory approach to offer the following advantages: (i) the definition of the problems to be solved comes from a direct understanding of the application environment, (ii) intermediate results can be applied to practical problems through the research process; feedback can be applied to future investigations, and (iii) the research process progressively becomes problem-focused as opposed to solution-driven.

Votta and Porter attempt to provide a credible model of empirical work for software engineering, something they feel is missing [Votta and Porter, 1995]. The authors discuss internal and external validity, the two factors upon which the degree of credibility depends upon. They then argue that not all studies have the same level of credibility, neither do they contribute the same depth or breadth of knowledge. This is illustrated in Figure 2.1 where the properties of different types of empirical work are plotted against the axes of dependence, validity, and test of theory. Anecdotal studies record what happened in a certain context at a certain time within a certain organisation. A case study goes somewhat further by attempting to show some correlation

between independent and dependent variables. Case studies are used when examining events within an organisation where the behaviour of the relevant variables cannot be manipulated. True experiments go even further and attempt to identify causality, i.e., provide a testable theory which explains why one event causes another. Experiments can only be used when the behaviour of the relevant variables can be manipulated.

There are also several other excellent papers which describe the concepts discussed above in a similar manner, e.g., [Brooks, 1980], [Kitchenham *et al.*, 1994], [Pfleeger, 1995].

2.2.1 Conclusions

Of the frameworks and methodologies reviewed, all contribute to the knowledge required to conduct empirical software engineering research. (It should be noted that these frameworks and methodologies are closely related to traditional approaches used within the behavioural and social sciences). None of the detailed frameworks or methodologies are superior given that they differ somewhat in the goals they try to achieve. The issues raised in [Moher and Schneider, 1982], [Basili *et al.*, 1986], and [Votta and Porter, 1995], however, are probably most relevant to the type of research conducted within this thesis.

2.3 Issues in experimentation

Given that empirical software engineering research is not an exact discipline, the planning, design, and analysis of an empirical study should be considered carefully. Several articles in the software engineering literature provide a good, general overview of these factors, e.g., see [Basili and Reiter, 1981], [Pfleeger, 1994], [Tiller, 1991], although they are not new concepts and, indeed, are very similar to those used within the behavioural sciences, e.g., see [Coolican, 1990], [Miller, 1975]. There are also other issues to be considered which are not widely reported in the software engineering literature: estimating the statistical power of an experiment, inductive analysis of the collected data, and confirmatory power (providing confidence in the findings of an experiment) through replication, and these are explained in this section.

2.3.1 Statistical significance testing

A common method of conducting an experiment is to use statistical significance testing of the Neyman-Pearson type: the form of rejecting or accepting a null hypothesis

(denoted H_0), where the null hypothesis is stated simply for the purpose that it may be rejected. The researcher then accepts the alternative hypothesis (denoted H_1) and concludes that an effect exists. For example, an experiment concerned with programmer productivity might have a null hypothesis

> H_0 : the mean programmer productivity of group A (treatment) is the same as that of group B (control)

with the alternative hypothesis stated as

> H_1 : the mean programmer productivity of group A (treatment) is greater than that of group B (control).

Once the researcher has stated the null and alternative hypotheses the significance criterion (α) should be set: α represents the chosen risk of committing a Type I error (incorrectly rejecting the true null hypothesis) and commonly quoted values within software engineering are 0.05 or 0.1. The empirical study is then conducted, the collected data analysed, and statistical tests applied. If the researcher achieves a statistical result that is less than the preset α value, the null hypothesis is rejected and the alternative hypothesis accepted.

From the many articles read by the author, it is clear that researchers within software engineering use this type of significance testing as their primary means to detect the presence of an effect within the phenomena being empirically investigated. An important but rarely considered part of significance testing is statistical power analysis. Statistical power analysis attempts to minimise the other error that can arise from significance testing: Type II error (incorrectly accepting the false null hypothesis).

2.3.2 Statistical power analysis

Statistical power analysis, an inherent part of significance testing, is not commonly presented in research findings. Sawyer and Ball define power to be the probability that a statistical test will correctly reject a false null hypothesis, i.e., the chance that if an effect exists it will be found [Sawyer and Ball, 1981]. For example, a power level of 0.4 means that if an experiment is run ten times, an existing effect will be discovered four times out of the ten experimental runs (the other six times a Type II error is committed). An adequate power level is usually quoted at 0.7 or 0.8, i.e., the chance an existing effect will not be detected is approximately one in five. Conducting a power analysis involves the following three components as well as the required power level:

The significance criterion (α): the chosen risk of committing a Type I error when performing significance testing. The directionality of the test (the test can be directional or non-directional) is also of importance. Power can be increased at the expense of a larger probability of committing a Type I error, e.g., raising α from 0.05 to 0.1, or by using a directional statistical test.

The sample size (N): the larger the number of subjects, the smaller the error, the greater the accuracy, and therefore the higher the power of the test.

The effect size (γ): the degree to which the phenomenon under study is present in the population. If all other factors are constant then the larger the effect size, the greater the probability the effect will be detected and the null hypothesis rejected.

The power level and these three determinants are related in such a manner that given any three values, the fourth can be calculated. Ideally, the researcher should estimate or anticipate the effect size, set the significance criterion, and specify the power level desired. The number of subjects needed to meet these specifications can then be estimated from the appropriate statistical tables, such as the ones presented in [Cohen, 1969]. The most difficult aspect of conducting a statistical power analysis is accurately estimating the effect size. In areas which have been empirically researched for some time there are usually studies from which an accurate estimate can be made, but in software engineering, due to a lack of empirical studies, this is not a viable option. The best method for a reasonable estimation may be to rely upon expert judgment: since experts have a realistic set of expectations about the effect of a particular phenomenon (within their field of expertise) this may be used to produce an effect size for the power calculation. For future studies, researchers can revise any effect size estimation by also taking in account the results of earlier experiments. (This and other concepts of statistical power are discussed in greater detail in [Miller *et al.*, 1995]).

Power of the statistical test becomes a particularly important factor when the null hypothesis cannot be rejected. The lower the power of the test the less likely it is the null hypothesis is accepted correctly. Consequently, when the null hypothesis is not rejected and the statistical test has low power, the only conclusion that can be made is that the effect examined has not been demonstrated by the study, not that the effect does not exist. Conversely, studies with a high power allow an interpretation of the results when there is insignificance because there exists strong support for the decision not to reject the null hypothesis, i.e., the chance that a Type II error has been made

is much smaller.

The importance of power is illustrated by means of the following example drawn from the medical literature: Robins was interested in determining why so much inconsistency existed in the specialised area of clinical depression [Robins, 1988]. Of the 87 studies Robins examined he found that only 8 of these had adequate power. The high power studies all reported a significant relationship, while the low power studies tended not to support the relationship. As a consequence, what seemed to be a large number of studies with inconsistent findings was actually only a small number of studies which provided consistent, meaningful, and reliable conclusions.

2.3.3 Inductive analysis

Quantitative empirical studies involving human subjects are usually best supplemented with more qualitative data gathering. This is because traditional statistical approaches strictly require identification of independent, dependent, and random variables with no errors of omission or commission. The assumptions underlying parametric statistics are often never tested (e.g., the assumption of normality). And statistical approaches alone tend not give the reader any real insight into the actual data. Gaining such insights is important for subject-based experiments as it is probable that more than one interpretation can be placed on the data. Statistical approaches, based solely on product measures of performance, do not readily facilitate investigation of such interpretations.

The inductive analysis paradigm is promoted by Brooks *et al.* who argue that it overcomes the weaknesses of traditional approaches through the production of facts and rules [Brooks *et al.*, 1987]. The paradigm recommends that data should be gathered from as natural setting as possible to allow comparison with real life environments. The result of the inductive analysis can then be used to supplement statistical meaning and to explain individual results. While simple facts can easily be obtained by inspection of the data, e.g., in an experiment involving maintenance tasks it should be easy to identify the number of subjects who made all modifications without any errors, rules governing potential cause and effects of the data are less easily extracted. Consequently, the paradigm advocates the use of automated techniques to accelerate the inductive analysis process and to ensure that potential rules are not overlooked. Brooks *et al.* have successfully applied the technique of inductive analysis to human computer interaction data [Brooks *et al.*, 1987].

An inductive analysis, which can manipulate both ordered and unordered variables, enables the experimenter to build a database of experimental results [Brooks and Vezza, 1989]. The rule induction software takes each database variable in turn as the dependent variable. Moreover, new variables, noted after the experiment, may be introduced to the database as long as variable values are obtainable. When a 'more planned' experiment is being designed, Brooks and Vezza are adamant that data gathering should remain as unrestricted as possible, thus allowing an inductive analysis to be performed, thus ensuring that any conflicting interpretations are considered [Brooks and Vezza, 1989]. It should be noted that this is not a shotgun correlation approach, an approach which attempts to correlate as many measures as possible against one or more dependent variables in the hope of finding a significant, but possibly meaningless, relationship [Courtney and Gustafson, 1993]. The inductive analysis paradigm examines individual datum points looking for trends between them that may present an alternative interpretation from any statistical analyses: the emphasis is on trying to understand what took place; it is more similar in concept to classification trees [Basili and Selby, 1991]. Classification trees, using software metrics to characterise software components, are the basis of predicting whether or not a particular component is likely to have some property, e.g., prone to a large number of errors.

The inductive analysis approach is especially applicable when conducting replications: if the results are different from that of the original study an inductive analysis may uncover trends within the data which offer interpretations as to why the difference was achieved (as Part II will show).

2.3.4 Replication

To provide confirmatory power all software engineering experiments should be reproducible [Brooks *et al.*, 1995]. Replication is vital to other disciplines such as the behavioural and social sciences; it provides either supporting evidence or questions the validity of the original experiment. By other researchers repeating an individual's empirical work confidence is built up in the procedure and the result. Without replication, a result should be at best regarded as of limited importance and at worst with suspicion and mistrust. Replication takes two forms: internal and external. Internal replication is undertaken by the original experimenters; external replication is undertaken by independent researchers and is critical for establishing sound results. Part II of this thesis illustrates the importance of external replication.

2.4 A critique of relevant empirical studies

From the previous sections it can be seen that the important features of performing empirical research are: (i) experimental design, (ii) the number of participating subjects, (iii) empirical data collection, (iv) analysis of the data, (v) the empirical detail that should be reported, and (vi) the conclusions drawn from the results. This section illustrates the effect of each of these features by highlighting the positive and negative aspects of the practices used by researchers in various selected studies. It is essential to realise the importance of ensuring that practices adopted for each of these features are appropriate; at the same time, what the impact will be if these practices are inappropriate. Note that the purpose of this review is to illustrate positive and negative aspects of these practices: it is not intended to be a definitive review of the empirical software engineering literature.

2.4.1 Experimental design

If the design of an experiment is poor or inappropriate it is likely to invalidate the results of the study. This section examines four separate experiments, two with designs that can be criticised for lack of planning and two with designs that can be considered effective for testing their stated hypotheses.

First, Mitchell *et al.* measured the effects of abstract data types (ADTs) on program development claiming that delaying the implementation details of ADTs until the coding phase lead to reduced programmer productivity, increased program inefficiency, and code quality which was no better than the quality of the code produced without ADTs [Mitchell *et al.*, 1987]. The experimental design can be criticised on two counts. (i) The dependent variable, source code quality, was measured by the number of lines of code, i.e., the more lines of code produced, the poorer the quality of the code. This variable cannot be regarded as an acceptable measure of code quality for the simple reason that it does not take account of the important factors which are used to measure quality, e.g., defect density, coupling, cohesion. Even worse, modularity of the code, which was also measured during the experiment, was not used as an indicator of quality. (ii) Productivity was measured as a combination of time logged into the system, the number of days to complete the design and implement it, and the average number of lines per hour. For this experiment, this cannot be regarded as a fair measure of productivity — subjects who had to use ADTs in their design had only one hour of instruction (and no training) on how to define ADTs in the

design notation without declaring the details, but had to present their design in the instructed manner; the other subjects had no such constraint. It is unsurprising to find that subjects using ADTs took longer to complete their design and implementation, and were subsequently classed as less productive. As a consequence, results have been produced that cannot be interpreted as anything other than meaningless.

Second, Lewis *et al.* empirically tested the relationship between software reuse and the object-oriented paradigm concluding that substantial productivity benefits are achieved over the procedural approach and that the object-oriented paradigm has a particular affinity for software reuse [Lewis *et al.*, 1992]. The experimental design can be criticised on three counts. (i) Productivity is not measured by total development time, an essential productivity measure. The reader has no idea if the object-oriented development took more or less time than the procedural development time. Consequently, the main measures of productivity (number of program runs, number of run time errors, and time to fix the run time errors) do not determine programmer productivity accurately enough to support the conclusion Lewis *et al.* make about it. (ii) The independent variable was the reuse of software components and subjects were divided into three categories according to the level of reuse — no reuse, moderate reuse (reuse as see fit), and strong reuse (reuse anything remotely appropriate). Arguably, this is an unnatural scenario, where no reuse and strong reuse subjects have unrealistic constraints placed upon them. (iii) No discussion of the effort required to produce the software components which were available for reuse takes places. Consequently, the results of this experiment can be best described as unconvincing, largely because an important variable, total development time, was not measured (Section 2.4.3 provides additional criticism of the data collection methods).

In contrast, an experiment which is planned and designed appropriately provides confidence that the data obtained is not influenced by some unforeseen factor. In their empirical study, Zweben *et al.* tested the effects of layering and encapsulation on software development costs and quality [Zweben *et al.*, 1995]. A set of three experiments compared the use of a layering approach when implementing new components (layering new components on top of existing components using only information about their functionality and interfaces) to a direct implementation approach (implementing new components from the coding details of existing components). Zweben *et al.* make careful selection of the independent variable (reuse approach — layering or direct) and the dependent variables (effort — measured by development time, component quality — measured by number of defects). The experimental subjects were part of a

university class on software components in Ada; all were graduate or upper-division undergraduate Computer Scientists. The subjects were randomly divided into two groups, counter-balanced by experience. Each group performed the task using both approaches, but in a different sequence (a sequence effect was catered for during analysis). Zweben *et al.* fully considered the random factors that could affect their independent variable, and designed their experiment accordingly. The only real cause for concern is the manner in which the experiments were conducted: the Ada class was run on two separate occasions, approximately 18 months apart. Unfortunately, the authors do not directly detail whether the three experiments were run for each class and the data grouped together or if one experiment was run for one class and two experiments for the other. The latter option (the one which is assumed to have been used) is fairly secure, but the previous option introduces potential variability and should really be reported as an internal replication. Apart from this oversight the authors report their design reasoning in considerable detail; the paper is an example of how to effectively design an experiment.

Another carefully considered experimental design is presented by Porter *et al.* who compare detection methods for software requirements inspections [Porter *et al.*, 1995]. Subjects for the experiment were members of a graduate course in formal methods. Independent and dependent variables were clearly identified and reported. Porter *et al.* evaluated the use of three different detection methods, but because one of these methods was more systematic than the others, concern was raised that subjects using this method first might distort the use of other methods in the second round. Consequently, a fractional factorial experimental design was discarded and the authors opted for a more appropriate partial factorial design — teams of subjects participated in two inspections using some combination of the three detection methods, but teams using the systematic approach in the first round also used it in the second round. To complicate matters the study was conducted over two runs six months apart. Unlike Zweben *et al.* this variable is specifically factored into the design and fully reported. At the same time, it allows criticism about allocation of subjects into teams and teams to detection methods: for the first experimental run teams of equal ability were created by rating subjects' background knowledge and experience and blocking subjects across teams, then allocating teams across detection methods; for the second experimental run, subjects were randomly assigned to teams which were then randomly assigned to detection methods. Porter *et al.* attempt to justify this strategy by stating that although teams from the first run may have had equal ability, in comparison teams for

subsequent runs would not. This is not a strong argument because using the random approach may have created teams of high and low ability by chance, thus introducing variability into the experiment. In addition, the random approach meant unequal numbers of teams were allocated to each detection method. The strategy employed should have been consistent, i.e., use random assignment for both runs or create teams of equal ability for both runs, either method provides less potential variability than the mixed approach used. Porter *et al.* designed a complex experiment, but one which is a good example of the numerous different variables that have to be considered when dealing with multiple treatments. The authors make a substantial effort to cater for random factors manipulating their dependent variables, but also demonstrate that the more complex the design, the easier it is to make an oversight.

2.4.2 Number of subjects

As discussed in Section 2.3.2 the larger the number of subjects participating in an experiment the higher the power level of the statistical test. So it is important to run the experiment with as many subjects as time and resources allow. There are various methods of recruiting subjects to participate in experiments, e.g., see [Keppel *et al.*, 1992]. Unfortunately, the most commonly used approach in software engineering appears to be availability sampling, i.e., use people that are available to participate as subjects. Consequently, there are many experiments conducted using only a handful of subjects, a situation which can cause problems during analysis. These problems are highlighted below.

First, Moreau and Dominick established a programming environment evaluation methodology for object-oriented systems and presented a test case application through an experiment using four subjects, concluding that the object-oriented software produced was more verbose, but programmer productivity was improved [Moreau and Dominick, 1990]. The primary focus of the experiment was the development time for C versus C++ for each subject across three tasks. For one task, however, a subject's performance was far poorer on the C++ development, a situation laconically explained as a debugging difficulty. In an experiment using a large number of subjects such an explanation may be sufficient. In a situation where this constitutes 25% of the subjects a full explanation should be provided.

Second, Zweben *et al.* used 10 subjects for two of of their three experiments testing the effects of layering and encapsulation on software development costs and quality [Zweben *et al.*, 1995]. Each experiment was a related design, but because sequence

effects were detected only 10 observations (5 per group) from 20 could be used during analysis. For the first of the two experiments, a significant effect was discovered for initial design and coding time, total development time, and the defect data collected. In contrast, the second experiment obtained significance for only the initial design and coding time, although all results were in the same direction. A possible reason for the difference in significance levels is because the power of the statistical tests was not high enough in the second experiment. If more subjects had been recruited, it is more likely statistical significance would have been achieved.

One caveat, using large numbers of subjects does not necessarily mean more reliable results. For example, in their experiment investigating the validity of cognitive fit[1] in the area of recursion and iteration, Sinha and Vessey recruited 82 subjects by paying each individual $10 to participate [Sinha and Vessey, 1992]. To motivate subjects to complete the task as quickly as possible cash prizes were awarded to the three fastest finishers of each group. Such motivation could have influenced the subjects' choice of construct (recursion or iteration) to solve the problems, subsequently distorting the results of the experiment.

As a consequence, the following general conclusions apply, (i) the smaller the sample size the more sensitive the experiment is to extreme performance from any individual subject, (ii) small subjects pools, because they are unlikely to be representative of any population, make it difficult to generalise the experimental results, (iii) the larger the number of subjects, the greater the power of a statistical test, (iv) experiments conducted using particularly small numbers of subjects should really only be a prelude to an investigation using a larger number of subjects, (v) whenever possible avoid the use of availability sampling as it can mean conducting an experiment with only a small number of subjects.

2.4.3 Data collection

Data collection is another of the crucial factors a researcher should be concerned with when conducting experimentation: if the data is not collected using accurate procedures or important measures are not collected at all, the data analysis and results are likely to be open to criticism. Reasons for directing such criticism are discussed below.

First, in their experiment, Lewis *et al.* asked subjects to record their own data

[1] Where the problem representation and problem solving tools and techniques support the strategies required to solve a given problem.

while participating in the experiment [Lewis *et al.*, 1992]. Lewis *et al.* argue this data collection method is valid by stating that the subjects (who were students) were promised anonymity from their data collection sheets and also that the data would have no bearing on their class evaluation. In contrast, subjects were warned if the data was not given accurately it would have a negative impact on their class evaluation. (How this would have been recognised from only anonymous data sheets is not discussed). Regardless of promises and warnings, the measures recorded, the number of program runs, the number of run time errors, time to fix the run time errors, the number of edits, and the number of syntax errors, are not measures that are amenable to accurate collection by hand, e.g., it is not difficult to imagine subjects forgetting to record how many times they ran their program. Consequently, the accuracy of the collected data can be questioned. Because the measures were taken by the subjects "some measures that might have been of interest were not collected" [Lewis *et al.*, 1992]. So a further criticism arises: total development time was not measured, arguably a more important measure for this experiment than any of the five that were taken. The validity of this reasoning should be severely questioned.

Second, in their experiment evaluating the maintainability of object-oriented software against procedural software, Henry *et al.* make use of two data collection mechanisms: the use of questionnaires and an automated data collection facility [Henry *et al.*, 1990]. Strangely, however, none of the quantitative variables, e.g., number of errors and total development time, were collected automatically. The subjects had to measure 17 such variables themselves by completing a questionnaire during the experiment, undoubtedly a difficult task to achieve when concentrating on solving the experimental problem. All the measures made by the subjects could have been taken automatically, something which would have eliminated variability caused by subjects taking their own measures and given the experiment more merit.

In contrast, it is more difficult to criticise the accuracy of automatic data collection procedures. For example, for their experiment on the performance and strategies of programmers new to object-oriented techniques, van Hillegersberg *et al.* built a special development environment to collect the majority of data automatically [van Hillegersberg *et al.*, 1995]. Using the development environment, subjects were able to edit, compile, and run the source code until all experimental tasks were completed. For each subject, the environment measured the amount of time taken to complete each task and their problem solving behaviour by registering how the subject navigated the source code, what parts of the program were changed, and what error messages were

received. In addition, a post-experiment questionnaire obtained subjects' views of task difficulty, motivation, and confidence. This paper provides an excellent example of how to collect data accurately and efficiently.

While automated data collection supplemented with debriefing questionnaires or interviews is probably the most accurate method of collecting data, Basili and Weiss demonstrate that data collection by hand can also be accurate, providing data validation takes place [Basili and Weiss, 1984]. In their experiment on classifying changes to program libraries, Basili and Weiss made each programmer who caused a change complete a change report form. Many of the reported changes were categorised incorrectly by programmers, a fact discovered by the data validation procedure — interviewing the programmer who completed the change report form. The data collection was carried out across three software projects; by completion of the data collection for the third project, Basili and Weiss estimated that at most 3% of completed forms were inaccurate.

Finally, other techniques such as video and audio recording have also been shown to be successful, e.g., see [Denning *et al.*, 1990], [Lee and Pennington, 1994].

2.4.4 Data analysis

Analysis of experimental data should be more than just application of statistical tests, it should also attempt to explain any subject variability and outlier data — especially if, as is commonly performed in software engineering experiments, outlier data are removed from the data set to which the statistical tests are applied. Four data analyses are reviewed to illustrate the difference between an insufficient and a thorough analysis.

First, in their experiment investigating the performance and strategies of programmers new to object-oriented techniques, van Hillegersberg *et al.* concluded that the hypothesis that object-oriented concepts are easy to learn and use did not hold for their experimental subjects [van Hillegersberg *et al.*, 1995]. The analysis is not comprehensive enough for several reasons. (A disappointing conclusion given the excellent data collection facilities used in the experiment as described in Section 2.4.3). (i) The conclusion is drawn because for six different experimental tasks, subjects maintaining the structured program were quicker to complete four out of the six tasks (three of these results achieved statistical significance) and were more productive (task score divided by completion time) for five out of the six tasks (four of these results achieved statistical significance). These are interesting results, but they need more explanation, i.e., the authors do not offer any reason as to why their object-oriented subjects took

longer and were less productive. The data recorded should have revealed such information, yet it is not discussed. (ii) The authors do not attempt to explain why the object-oriented subjects completed two of the tasks quicker than structured subjects, nor why they were more productive for one of these tasks. While these results were not statistically significant, because they were in the opposite direction to the rest of the results, explanations should have been sought. Did these tasks simply facilitate an object-oriented solution or were there other factors involved? (iii) The problem solving behaviour of each subject was recorded, but no analysis of this data take place. As a consequence, the results of an experiment investigating an important area within the object-oriented paradigm are not as comprehensive or interesting as they should be.

Second, Lohse and Zweben performed a series of experiments investigating the effect of module coupling on system modifiability [Lohse and Zweben, 1984]. The experiments compared the effect of modifications on two versions of the same system, one where all procedure communication was via parameters, the other where all parameters were replaced by global variables. Statistical tests did not find any significant effect due to coupling type — no further analysis is undertaken. Lohse and Zweben do not attempt to explain their data or what problem solving processes subjects were using. The authors merely conclude that the results of the experiment call into question the effect of parameters versus globals on system modifiability, something which they cannot do because the power of their statistical tests was not considered. Lohse and Zweben should have performed a more thorough analysis of the data, although the data collected was probably insufficient to allow this. This experiment demonstrates that if the appropriate data is not collected, an inadequate analysis will follow.

In contrast, thorough data analysis inspires results which are more meaningful. For example, Basili and Reiter conducted a controlled experiment to investigate the the effects of a disciplined methodology on software development [Basili and Reiter, 1981]. The data analysis of this experiment was thorough and its findings clearly reported. The authors identify and justify their use of statistical tests and fully report the results of their application. An impact analysis of these statistical results takes place, the authors arguing that strong statistical impact is demonstrated by an actual rejection percentage of null hypotheses well above the expected (by chance) rejection percentage. Basili and Reiter then clearly and concisely describe the results of their statistical analysis by itemising the individual differences identified by the study. Finally, the authors evaluate these differences based upon a general set of beliefs regarding software development which is then used to strongly substantiate the claims

that (i) methodological discipline is a key influence on the efficiency of the software development process and (ii) disciplined methodology significantly reduces the costs of software development. The experiment provides an excellent example of how to carefully plan, rigorously perform, and report data analysis.

Similarly, in their laboratory experiment, Frakes and Pole investigated the effects of four different representation methods on the ability to search a database for reusable components [Frakes and Pole, 1994]. The analysis begins by evaluating the searching effectiveness of each method with recall (the number of relevant items retrieved over the total number of relevant items in the database), precision (the number of relevant items retrieved over the total number of relevant items), and overlap (the ratio of the number of relevant items in the intersection of two methods over the number of relevant items in their union). The authors are very thorough in their reporting of this part of the analysis supplying the full details of their ANOVA tests as well indicating the subject variability for recall and precision for each method via boxplots. Frakes and Pole also evaluate the search time for the four representation methods (some were found to be significantly faster to use than others) and indicate subject variability via boxplots. To check for an ability effect, UNIX and programming experience were correlated against search effectiveness and search time — correlation figures were presented, but no significant results were achieved. Finally, to gain additional insight into the usefulness of the methods, subjects were debriefed on two issues: (i) their preference of methods — ranking each method in order of preference and the rating of each method on a seven point scale of usefulness, and (ii) the helpfulness of methods for understanding the software components examined — each method was rated on a seven point scale. The information gained from this useful debriefing is presented in tables and boxplots. As a consequence, this experiment provides the reader with meaningful and fully comprehensible data. The conclusions drawn are self-explanatory given the thorough data analysis Frakes and Pole conducted.

2.4.5 Reported detail

For an experiment to be fully appreciated the written report should detail as much accurate information as possible. Missing details can reduce the importance the experimental results might have; worse, missing details can actually leave readers skeptical of the results. Furthermore, it makes external replication extremely difficult, if not impossible. Two examples are used to illustrate this point.

First, Mancl and Havanas report on a case study of the impact of object-oriented

software on maintenance [Mancl and Havanas, 1990]. The study is based on a system which is part developed using structured techniques and part developed using object-oriented techniques. Mancl and Havanas compared subsystems developed using these techniques, concluding that the measurements taken show an increase in software reuse, increased programmer productivity, and reduced complexity of software changes for the object-oriented developed subsystems. Unfortunately, the findings are confused by too many missing details. (i) The authors do not explain what type of subsystems are developed using structured techniques and what type of subsystems are developed using object-oriented techniques. Were the object-oriented subsystems implementing less complex functionality? (ii) Many of the object-oriented subsystems used in the comparison were first designed using structured techniques. How much effort went into redesigning these subsystems is not detailed, e.g., were they designed for reuse and ease of maintenance where the original designs were not? (Presumably if they had, then the redesign should not have been necessary). (iii) It is not reported what the criteria was for redesigning a structured subsystem. It could be that all the smaller less complex subsystems were redesigned, but larger, more complex ones were not because of the investment required to do so. As a consequence, the missing details make the reader wonder if the subsystems compared by Mancl and Havanas are actually comparable, thus questioning the value of conclusions.

Second, in their experiment evaluating the maintainability of object-oriented software against procedural software Henry *et al.* do not report enough detail on the collected data to allow sensible conclusions to be drawn [Henry *et al.*, 1990]. (i) Subjects were divided into two groups, one group solving the experimental tasks using the object-oriented software then the procedural software, the other group doing the reverse. The data for these two groups is presented in such a manner that only the mean totals for each group is given, i.e., there is no data which shows how subjects performed on the object-oriented and the procedural software, only the accumulation of the two performances is given. On its own, this data is almost meaningless. (ii) There is no indication of how individual subjects performed in the experiment, nor is there any indication of subject variability (an essential factor to be considered given the data collection methods used, see Section 2.4.3). It is unknown if the experimental results were adversely affected by such factors. (iii) Henry *et al.* conclude that object-oriented software is more maintainable based on the fact that it requires fewer modules to be edited, fewer sections to be edited, and fewer lines of code to be modified or added. Subjects maintaining the object-oriented software first, however, perceived

the experimental tasks to be more difficult than subjects maintaining the procedural software first. It is unknown if total development time also reflected this finding, i.e., did subjects actually take longer to maintain the object-oriented software? Henry *et al.* have too many missing details from their collected data to allow their experimental results to be placed into any context. Furthermore, a personal request for the missing information was turned down on the basis that it was not available.

Part II of this thesis details the lessons learned from performing an external replication and introduces guidelines regarding the required level of reported detail for software engineering experiments to be fully appreciated and to allow other researchers to attempt an external replication.

2.4.6 Experimental results and conclusions

Once the experimental process has been completed, it is important not to overstate the results of the experiment, especially if the experiment is the first to test a particular hypothesis. (Results emerging from a programme of research or from an external replication should allow stronger conclusions to be drawn). The main difficulty is the ability to generalise the results of the experiment to other samples of the population and to extrapolate the results from the laboratory to industrial practice. Factors which limit any generalisation are termed threats to external validity. Ideally, researchers should identify what they view as the threats to their experiment's external validity and report the requirements to address such threats. Unfortunately, this is not commonly performed in practice. One example is provided in [Porter *et al.*, 1995] for their experiment comparing detection methods for software requirement inspections. The authors identify three threats to external validity: (i) subjects may not be representative of software professionals, (ii) the specification documents used may not be representative of real programming problems, and (iii) the inspection process used many not be representative of software development practice. The authors conclude that surmounting these threats requires experimental replication using software professionals to inspect industrial work products. The information provided by Porter *et al.* allows their results to be placed in context; in future, researchers should look to repeat this practice.

2.4.7 Conclusions from critique

The empirical literature reviewed has highlighted examples of contrasting practices on important experimental issues. The most important conclusion derived from these

examples is that empirically-based research does not necessarily mean well performed research: it has been demonstrated that experiments can produce results that are meaningless. Fenton *et al.* support this view and provide five definitive questions which should be asked about any claim made of software engineering research [Fenton *et al.*, 1994]:

1. Is it based on empirical evaluation or data?
2. Was the experiment designed correctly?
3. Is it based on a toy or real situation?
4. Were the measurements used appropriate?
5. Was the experiment run over a sufficient period of time?

Although all the answers to these five questions are positive for very few existing empirical studies, and it is unrealistic to expect the answers to be positive for all five questions for future empirical studies, such questions do enable other researchers to gauge the worth and importance of an individual piece of empirical research. In future, the questions asked by Fenton *et al.*, along with the practices highlighted in this review, may help researchers to focus on improving any empirical work they perform.

This review has also highlighted certain deficiencies in software engineering empirical practice, namely the lack of external replication and the lack of results emerging from a programme of empirical research. Votta and Porter agree that deficiencies exist and present advice on what the software engineering community should undertake to improve the field's ability to support credible empirical work [Votta and Porter, 1995]:

1. Accept the need to repeat experiments and publish the results of the replicated experiments whether they agree or disagree with the results of the original.
2. Encourage the search for explanations when the results of replicated experiments differ.
3. Sets of theories which are not testable empirically should not be allowed.
4. Models of the differences between student and professional developers in industrial settings must be constructed so that the validity of student studies can be understood.
5. Cheaper methods of experimentation should be explored, e.g., simulation, efficient data collection.

6. More access to real project data is required.

7. Empirical work must be recognised as important and necessary for a successful discipline.

Whether or not all these 'next steps' are achievable is debatable, but Votta and Porter identify points which should be seriously considered.

To conclude, researchers should be more aware that there is much that can be improved in empirical software engineering practice. They should also be aware that to make empirical results more reliable and generalisable there is a need for external replication and programmes of empirical research to be conducted.

2.5 Summary

This chapter has introduced the various frameworks and methodologies for conducting empirical software engineering research; it has summarised the issues that need to be considered when conducting empirical research, but which are rarely reported in the literature; finally, it has reviewed and examined important examples of good and bad practice within software engineering empirical research and has highlighted the major deficiencies in current practices. The chapters that follow build upon this review by emphasizing the importance of external replication and introducing a methodology termed the multi-method approach.

Part II

REPLICATION

Chapter 3

Confirmatory Power Through Replication

3.1 Introduction

Every stage of an empirical study from background reading to result interpretation to writing the report and the conclusions is prone to error. These are but some of the problems: (i) The background may not be properly researched and the study may be addressing an unimportant issue. (ii) Inappropriate methods may be used. For example, with subject-based experiments, strictly controlled laboratory experiments are usually best supplemented with more qualitative forms of data gathering. Also, an insufficient sample size can lead to a situation where a real, even quite sizable, effect has little chance of being revealed as significant. (iii) Errors of commission or omission may be made or experimental variables may be incorrectly classified. (iv) Statistical procedures may be misapplied. For example, parametric statistics may be applied to non-normal data. (v) Alternative interpretations may not be presented. With subject-based experimentation, it is probable that more than one interpretation can be placed on the data. (vi) The experimental methods may be poorly reported so that it is impossible to perform an external replication of the study. For example, instructions and task materials given to subjects may not be fully reported or may otherwise be unobtainable. Other researchers, for example, [Basili *et al.*, 1986], [MacDonell, 1991], have already criticised empirical studies within the software engineering literature for poor reporting. (vii) Conclusions may not be justified from the analysis or may simply

be incorrectly expressed in terms of the null hypothesis. As a result, there are dangers in drawing conclusions from a single empirical study which has no confirmatory power.

Consequently, scientists demand that empirical results are reproducible and, importantly, externally reproducible, i.e., an independent group of researchers can replicate the experiment and obtain similar results. Results of a replication can provide either supporting evidence or question the claims arising from the original experiment. By other researchers repeating an individual's empirical results, confidence is built up in both the procedure and the original claims. Without the confirming power of a replication, empirically-based claims in software engineering should be regarded at best as of limited value and at worst with suspicion and mistrust. Replication takes two forms: internal and external. Internal replications are undertaken by the original experimenters to increase their own confidence in the results; external replications are undertaken by independent researchers who seek to check and improve on the findings of other researchers and are critical for establishing sound results.

This chapter introduces and expands upon the concept of external replication and details its importance for producing more reliable and generalisable conclusions within software engineering (making use of the work contained within [Brooks *et al.*, 1995]). Chapter 4 presents the external replication of a software engineering experiment. The replication illustrates how easily one or two uncontrolled variables can influence the results of an experiment and subsequently demonstrates the importance of performing external replications. Chapter 5 discusses the lessons learned and presents guidance on the required level of reported detail to allow others to conduct an external replication.

3.2 The ideology of replication

Subjecting theory to experimental test is a crucial scientific activity. Popper, however, explains that researchers must be sure of their results before reporting them, stating

> We do not take even our own observations quite seriously, or accept them as scientific observation, until we have repeated and tested them [Popper, 1968].

Goldstein and Goldstein take this one step further, stating

> We now take for granted that any observation, any determination of a 'fact', even if made by a reputable and competent scientist, might be doubted. It may be necessary to repeat an observation to confirm or reject it. Science

> is thus limited to what we might call 'public' facts. Anybody must be able to check them; experimental observations must be *repeatable* [Goldstein and Goldstein, 1978].

Researchers within the software engineering community are beginning to take this advice seriously and are demanding that experimental results are replicable by an external agency. For example, Lewis *et al.* claim

> The use of precise, repeatable experiments is the hallmark of a mature scientific or engineering discipline [Lewis *et al.*, 1991].

And the greater the number of external replications the better, at least until such time additional replications carry no further confirmatory power. The author agrees with Curtis who argues that results are more impressive when they emerge from a programme of research rather than from one-shot studies (although Curtis refers to internal replications and does not mention the greater confirmatory power of external replications) [Curtis, 1980].

Broad and Wade, in their description of the scientific ideology, consider replication to be the third check in verifying scientific claims, the first two being the peer review system that awards research grants and the journal refereeing that takes place before publication. The authors also describe the ideal of reporting experiments as follows

> A scientist who claims a new discovery must do so in such a way that others can verify the claim. Thus in describing an experiment a researcher will list the type of equipment used and the procedure followed, much like a chef's recipe. The more important the new discovery, the sooner researchers will try to replicate it in their own laboratories [Broad and Wade, 1986].

Replication is also concerned with the way the original hypothesis is expressed. According to Smith

> Replication does two things: first, it tests the linguistic formulation of the hypothesis; second, it tests the sufficiency of the explicit conditions for the occurrence of the phenomena [Smith, 1983].

One need not go as far as performing a replication to discover problems with the linguistic formulation of a hypothesis. For example, Henry *et al.* state in their conclusions that "the experiment supports the hypothesis that subjects produce more maintainable code with an object-oriented language than with a procedure-oriented

language" when in fact their subjects were asked to make modifications to an object-oriented system and a functionally equivalent procedure-oriented system which were originally developed by a single graduate student (a correct statement of the hypothesis is given earlier in their paper) [Henry *et al.*, 1990]. Regarding the sufficiency of the explicit conditions, an example would be where criteria for subject participation in a software engineering experiment is insufficiently specific. As a result, the replication yields different results owing to variability amongst subjects.

In conclusion, a stronger scientific and engineering foundation can only come about if the software engineering community becomes more involved in external replications.

3.2.1 Frequency of replication studies

A search of the empirical software literature was unable to find examples of external replication studies. Two recent software engineering reference works do not discuss external replication nor cite examples [McDermid, 1994], [Marciniak, 1994]. Even reviews of software testing and software maintenance where empirical research is more common make no mention of external replication work, e.g., [Roper, 1992], [Sharpe *et al.*, 1991].

A particular study which should have been externally replicated soon after its results were published was by Shneiderman *et al.* whose study called into question the utility of flowcharts in comparison to pseudocode [Shneiderman *et al.*, 1977]. According to Scanlan, this research led to the decline of flowcharts as a way to represent algorithms [Scanlan, 1989]. A review of their empirical study, however, lead Scanlan to believe that Shneiderman *et al.* had failed to consider key dependent variables in their experimental design, e.g., time to comprehend the documents (the subjects were all given as much time as they required). Scanlan designed a new experiment using time as a dependent measure and found that significantly less time was required to comprehend algorithms represented as flowcharts.[1] Scanlan's criticisms of Shneiderman *et al.* are deemed controversial, but he has demonstrated the ease with which the software engineering community may be mislead by accepting the results of a single empirical study without any confirmatory power. External replications must be conducted to confirm the results. The major concern, therefore, is the rarity of external replication in software engineering research.

[1] Scanlan's experiment is not an external replication because he designed a new experiment — only the hypothesis being tested was the same. See Section 3.3 for explanation of what constitutes an external replication.

3.2.2 Recipe improving

The amount of detail that can be used to describe the setting of an empirical study is without end. As a consequence, it is more correct to consider partial replication and the goal of performing as near exact replication as possible. According to Broad and Wade, exact replication is an impractical undertaking because the recipe of methods is incompletely reported, because to do so is very resource intensive, and because credit in science is won by performing original work. The authors do, however, draw attention to the important activity of improving upon experiments. They state,

> Scientists repeat the experiments of their rivals and colleagues, by and large, as ambitious cooks repeat recipes — for the purpose of *improving* them ... All will be adaptations or improvements or extensions. It is in this recipe-improvement process that an experiment is corroborated [Broad and Wade, 1986].

Baroudi and Orlikowski qualify this and note

> Where a study fails to reject a null hypothesis due to low power, conclusions about the phenomenon are not possible. Replications of the study, with greater power, may resolve the indeterminacy [Baroudi and Orlikowski, 1989].

3.2.3 Experimenters' regress

As noted by Collins, a paradox exists for those who want to use replication as a test of the truth of scientific claims:

> The problem is that, since experimentation is a matter of skillful practice, it can never be clear whether a second experiment has been done sufficiently well to count as a check on the results of the first. Some further test is needed to test the quality of the experiment — and so forth [Collins, 1985].

Although the external replication study presented in the following chapter failed to replicate the original experiment's results, it is shown how an inductive analysis of the data established possible interpretations of the differences between the replication and the original results. The inductive analysis, by yielding an understanding of the processes involved, resolved the paradox of the experimenters' regress.

3.3 A framework for replication

Internal replication is where the original experimenters have carried out a series of repeated experiments which may be published as one paper or as a series. Internal replications have some confirmatory power but it is less than that achieved by external replications. External replication means published experiments which are repeated by experimenters who are independent of those who originally carried out the empirical work. Exact replication is unattainable, so it is important to consider and categorise the difference.

Researchers undertaking an external replication must consider three things [Brooks *et al.*, 1995]. First, researchers must consider the experimental method and decide whether a similar or alternative method is to be used. According to Brewer and Hunter, the employment of multiple research methods adds to the strength of the evidence [Brewer and Hunter, 1989]. That is, a basic finding replicated over several different methods carries greater weight. Another option is to improve the existing experimental method, e.g., by debriefing subjects after the formal experiment is over. Such debriefings can provide insights into the processes involved. This type of improvement does not compromise the integrity of the replication.

Second, researchers must consider the experimental task(s) and decide whether a similar or alternative task is to be used. According to Curtis, when a finding can be repeated over different tasks it becomes more convincing [Curtis, 1980]. That is, a basic finding replicated over several different tasks carries greater weight. Another option is to improve the existing experimental task, e.g., by making it more realistic.

Third, researchers must consider the experimental subjects and decide whether a similar or alternative group of subjects are to be used. A basic finding replicated over several different categories of subjects carries greater weight. Another option is to improve the group of subjects, e.g., by using more subjects or more stringent criteria for participation.

The experimentation framework paradigm for software engineering has been established [Basili *et al.*, 1986] (and summarised in Chapter 2, Section 2.2), but should be extended to incorporate external replication and its various forms. The extension considers the three categories: (i) method, (ii) task, and (iii) subjects where each can be either `similar`, `alternative`, or `improved` (at this stage it is unnecessary to work with more detailed categorisations). These categories of method, task, and subjects could also be applied to internal replications. If the planned replication of an empirical

study is the first to be conducted then it may be more important to attempt to confirm the original results by keeping method, task, and subjects as similar to the original as possible; improvements and alternatives can be considered in later replications. Of course, if too many alternatives are used or if the scale of any recipe-improving is too substantial, it becomes debatable whether the study counts as a replication.

3.4 Summary

This chapter has introduced and expanded the concept of external replication and how it can provide confirmatory power for an empirical study. In addition, it has been argued that the experimental framework paradigm for software engineering should be explicitly extended to include external replications.

The following chapter presents an external replication study of an experiment which tested the effect of modularity on software maintenance.

Chapter 4

An External Replication Study

4.1 Introduction

It has been argued that external replication should have an important role in empirical software engineering research. A literature search failed to reveal any published examples of such work. Two major reasons existed for conducting the first external replication. First, the software engineering community had to be shown the value of conducting external replications and the only method of achieving this was to actually replicate an empirical study. Second, the author was not familiar with conducting subject-based experiments and wanted to experience the difficulties of such work; performing a replication is an ideal initiation. Once it had been decided to undertake an external replication, several weeks were spent reviewing a number of doctoral dissertations in the computing field where the emphasis was on empirical investigations. Korson's empirical study was concerned with the hypothesis that modular programs are easier and faster to maintain than non-modular ones under the condition that modularity has been used to implement information hiding which localises changes required by a modification [Korson, 1986], [Korson and Vaishnavi, 1986]. This study was chosen on several counts.

1. There exists uncertainty about the empirical foundation of the guideline that modular software is better, e.g., Fenton cites relatively early work which claims to show no strong evidence to support the widely held beliefs about the benefits of modularity [Fenton, 1994]. Korson's work on the other hand, claimed to show a large positive effect for modularity and seemed to be one of the better

performed and reported experiments: the raw data and histogram plots alone are enough to convince the reader of the result without resorting to the use of statistics. Consequently, it was unnecessary to perform a formal, statistical power analysis to check that there was a reasonable probability of detecting a significant result in the replication study.

2. Korson's thesis appeared to contain sufficiently detailed descriptions of experimental materials and procedures to allow a replication attempt. The thesis appendices contained complete code listings and lists of instructions given to subjects and the thesis body provided criteria for subject participation and a description of the experimental design and processes.

3. Korson claimed to have succeeded in providing internal replicability stating,

   > ... the study has demonstrated that a carefully designed empirical study using programmers can lead to replicable, unambiguous conclusions

4. On a first reading, Korson's work qualified as well performed empirical work. Also, no criticisms of it could be found in the literature, e.g., Banker *et al.* summarise Korson's findings without criticism [Banker *et al.*, 1993].

5. If Korson's work bears up to detailed scrutiny and is externally replicable, then this would increase confidence in the commonly accepted guideline that modular programs are easier to maintain than non-modular ones.

4.2 Review and critique of Korson's work

4.2.1 Review

Korson designed a series of four experiments each testing some aspect of maintenance. The experiment of most interest from this series sought to demonstrate the value of information hiding: it was designed to test if a modular program used to implement information hiding is faster to modify than a non-modular, but functionally equivalent version of the same program. The non-modular (or monolithic) program was created by replacing every procedure and function call in the modular version with the body of that procedure or function. Programmers were asked to make functionally equivalent changes to an inventory, point of sale program — either the modular version (approximately 1000 lines long) or the monolithic version (approximately 1400 lines long). The

maintenance tasks can be classified as perfective maintenance [Marciniak, 1994] — in this case the tasks were enhancing existing features of a program. Korson hypothesized that the time taken to make the maintenance changes would be significantly faster for the modular version. The modification process the subjects participated in contained four phases:

Think: on paper, the modifications are coded noting deletions, additions and changes to the original source code.

Edit: at the computer, the original program is edited to reflect the modifications made on paper.

Syntax: the syntax errors are interactively removed from the modifications.

Logic: logic errors are interactively debugged until the program passes a standard test.

Beforehand, participants were given ample time to read the instructions and to ask questions on the experimental process and the maintenance tasks. When subjects were satisfied that they understood the nature of the required modifications, each participant was given the appropriate program listing (modular or monolithic) and Phase 1 timings started. Phase 2 began when the participant had made the changes on paper and was ready to modify the actual program. Phase 3 started when the required changes had been entered but syntax errors had yet to be removed. Finally, Phase 4 was entered when all syntax errors had been removed. The times for each of these phases were recorded and used as a basis for the statistical results (see Section 4.5.1), but note that Korson regarded Phase 2 as an exercise in typing and does not present timing data for this phase nor involve this phase in the analysis.

Prior to the experiment each subject completed a pretest. This was used simply as a way of familiarising the subjects with the experimental design and the computer hardware and software environments. Korson summarised the results for his four experiments as follows:

> The study provides strong evidence that a modular program is faster to modify than a non-modular, but otherwise equivalent version of the same program, when the following conditions hold:
>
> (a) Modularity has been used to implement "information hiding" which localises changes required by a modification.

(b) Existing modules in a program perform useful generic operations, some of which can be used in implementing a modification.

(c) A significant understanding of, and changes to, the existing code are required for performing a modification.

In contrast, the study provides evidence that modifications not fitting into the above categories are unaided by the presence of modularity in the source code.

Condition (a) is the condition of interest here. Korson's result summary is given in full to draw attention to his claim that not all modifications are necessarily made more quickly by using modularised code.

4.2.2 Critique

On a first reading, Korson's work could be classified as well performed empirical work. On closer inspection, however, a number of weakness were highlighted.

- In the modular program version the global array variable `InventoryArray` is used instead of having a driver procedure with a local array variable formally passed as a parameter to the four modules which require access to it. Hence, the modular program does not fully implement information hiding and the result is biased: had parameter passing been implemented for the array variable, modular subjects may have taken longer to perform the maintenance task. Korson discusses how common coupling is allowed for an information cluster but the implementation allows the whole program access to the array variable rather than restricting access to the four procedures which might be construed as making up an information cluster.

- A subsequent criticism can be made because of the global declaration of the variable `InventoryArray`. Modular subjects are specifically asked to delete the declaration which is immediately succeeded by the following very helpful comment:

  {All access to inventory information is via the following 4 procedures}

  In the act of deleting the global array variable declaration, modular subjects, by reading the next line, are given the biased information that they need only look at the next four modules which are conveniently placed at the top of the

procedure section. Had there been a driver procedure with a local array variable, modular subjects would have been placed in the same position as the monolithic subjects in having first to take time to locate the relevant code. This criticism is all the more pertinent because in his thesis, Korson develops the explanation that linear searching of monolithic code is very time consuming while modular code allows the programmer to use a tree search strategy. Thus Korson incorrectly asserted that use of comments was consistent across the two program versions.

- The instruction to delete a global variable declaration is questionable from an entirely different point of view: a wary subject may suspect that such a variable is inadvertently used elsewhere in the program and so perform additional searches to check for references.

- A weakness of any piece of empirical work involving human subjects is the failure to supplement a traditional statistical approach with a more qualitative or ethnographic approach which seeks to provide alternative interpretations of the data at the level of individual subject behaviour. Korson relies solely on timing data in developing an interpretation of the data. While product measures of performance are important, some understanding of the underlying processes can dramatically change the interpretation of the results. Alternative interpretations may have been realised if notes made by subjects on program listings had been analysed; if subjects had been debriefed as to the cause of difficulties which resulted in highly skewed timings; if analysis had been performed on the various saved versions of the modified program during the phases of the experiment; or if analysis had been made of any differences between subjects who were professional programmers and those who were advanced computer science students.

- Another weakness of any piece of empirical work involving human subjects is a lack of realism in either the tasks or the context in which subjects are asked to perform the tasks. Korson's experiment can be regarded as having imposed an unrealistic ordering of the four phases. The weakness is that the most natural way for a programmer to locate code to change is by using the search facility of a text editor: in the Korson experiment, subjects had to manually search and write changes on a hardcopy listing of the program during Phase 1.

In later sections, evidence will be presented which corroborates these weaknesses.

4.3 A pilot study

4.3.1 Introduction

An initial pilot study was conducted with four experienced programmers to ensure that there were no problems with the experimental materials or the execution of the experimental tasks within the PC environment used. Problems that were discovered are now discussed. Comments made by the pilot subjects during an informal debriefing session were also noted and are presented in Appendix E.

4.3.2 Problems encountered during the pilot study

The pilot study session consisted of performing the required modifications to both the pretest and experiment programs. No problems were encountered during the pretest pilot — it was a straight forward modification to a page of Turbo Pascal code which sorted words into alphabetic order. During the experiment pilot, however, the following problems were encountered:

1. The Alt and function keys used for testing the program worked before the modification but failed after it because of performance changes not taken into consideration. This problem was later remedied for the experiment proper using a hot key customisation tool.

2. Some spelling mistakes in the documentation were noted and remedied for the experiment proper.

3. Subjects were unfamiliar with some of the American-specific terminology.

4. One of the subjects asked what the name of the file to modify was despite it being mentioned in the instructions.

5. A large sheet of test results caused data identification problems for some of the subjects.

The solutions adopted for the last three of these problems are discussed in the section on experimental design differences (Section 4.4.1).

4.4 The replication study

4.4.1 Experimental design differences

Several changes were made to the experimental instructions in an attempt to improve their readability and highlight particular points. These were minor recipe-improving changes aimed at reducing sources of variability.

As noted earlier, a problem was encountered with American-specific terminology, e.g., layaway. The solution adopted was to create explanatory footnotes in the documentation given to experimental subjects. To avoid subjects experiencing problems locating the name of the file to modify, the name of the file was highlighted in bold in the documentation (the filename was in parentheses in the middle of a large paragraph and was deemed to be easily missed on a first reading). Similarly, in the testing phase of the instructions the Alt key numbers were highlighted by placing in bold. The participants were given a listing of the input data file and a listing of the modified file after the program was run with test data for comparison with their actual program output. As with Korson, the actual data changes they were looking for on the listings were circled in order to reduce possible variability in testing times: this had been overlooked for the pilot study.

Korson made use of human monitors to record start and stop times for each of the experimental phases. In addition, monitors were present to counter any irregular circumstances which arose during the experiment, e.g., computer malfunction. As Korson did not record the number of participants to a monitor no comments can be made about differences. In this study there were four or five participants to each monitor.

Korson provided a Turbo Pascal reference manual for each participant. In this study only one such manual was provided between four or five participants although it was noted that no reference conflicts were caused by this limitation. Furthermore, Korson allowed his subjects to keep the Turbo Pascal reference manuals as payment for their participation. In this study the participating undergraduate student subjects were not charged for photocopied lecture notes later in the semester.

The replication study was run in the afternoon from 2pm until 6pm rather than between 6pm and 10pm as Korson had done. Both the pretest and experiment were run during this time. It is unclear if Korson also ran his pretest on the same occasion as his first experiment, although it appears unlikely. Between the pretest and the experiment the replication subjects were provided with a break and light refreshments.

4.4.2 Replication methodology

To achieve equivalent statistical power the replication required at least the number of subjects (16) that Korson used in the original experiment. Data for 17 out of 23 voluntary subjects was collected. Surprisingly, especially so given the comment of one the pilot subjects who felt that to complete the experimental task required no understanding of the software, six subjects (three from the modular group and three from the monolithic group) failed to complete the experiment by 6pm. Since no direct payment was made, in contrast to Korson, these subjects were not asked to return later to complete the experiment. The subjects, all members of the Computer Science Department at the University of Strathclyde, were a mixture of second year (5), third year (2), and final year undergraduate students (9), research students (4), and research assistants (3). While this could be said to be a fairly heterogeneous group in terms of age, qualifications and ability, all subjects met the criteria set by Korson: fluency in Pascal, knowledge of the IBM-PC, and an amount of programming experience. Subjects, randomly assigned to two groups, participated in the experiment in a single laboratory — subjects with monolithic programs sat next to subjects with modular programs to discourage plagiarism, although this was not a significant worry. Subjects were not told about the nature of the experiment, but were informed that different versions of the program existed and that, subsequently, certain subjects were likely to finish quicker than others. This was to reduce subjects' concern about their performance. The procedure each subject followed was:

1. pretest to familiarise subjects with the environment, e.g., the experimental procedure, the editor.

2. short break (10 - 15 minutes) for refreshments and to allow subjects to clear their thoughts,

3. experiment,

4. finally, each subject was asked for brief personal details and to complete a debriefing questionnaire (a copy of which can be found in Appendix E).

Both pretest and experiment were of the same format that Korson used (see Section 4.2) and the interested reader is directed to his thesis for details of the instructions, documentation, modification specifications, and full source code listings.

	Group 1 (Modular)					Group 2 (Monolithic)			
	think	syntax	logic	total		think	syntax	logic	total
Subject A	15	1	2	18	Subject I	15	4	5	24
Subject B	14	7	4	25	Subject J	32	6	3	41
Subject C	16	1	10	27	Subject K	38	2	4	44
Subject D	22	1	20	43	Subject L	44	1	4	49
Subject E	39	3	14	56	Subject M	28	11	11	50
Subject F	34	19	5	58	Subject N	36	22	1	59
Subject G	31	29	1	61	Subject O	29	3	29	61
Subject H	46	25	25	96	Subject P	55	2	36	93
					Subject Q	38	15	58	111
Mean(mins.)	27.1	10.8	10.1	48.0		35.0	7.3	16.8	59.1
(Korson	14.9	2.9	1.5	19.3		51.9	18.9	15.1	85.9)

Table 4.1: Result times of the replication study

4.5 Experimental results

4.5.1 Statistical results

The timing data collected during the experiment is presented in Table 4.1 (the results in parentheses are Korson's mean times) and the bar charts in Figure 4.1 display the mean modular times versus the mean monolithic times for Korson's times and the replication times. Given the size of the effect Korson discovered, it was hypothesized that the replication's results would be similar to Korson's, i.e., that the times for modular subjects would be significantly faster than for the monolithic subjects. Surprisingly, however, this did not turn out to be the case: mean syntax time actually took longer for the modular version. Furthermore, the difference between the modular and monolithic mean total times is relatively small: 48 minutes compared to 59.1 minutes. In contrast, Korson's times were 19.3 minutes compared to 85.9 minutes. On average, Korson's monolithic subjects took more than 4 times as long as his modular subjects yet the replication monolithic subjects took only approximately 1.3 times as long as the modular subjects. As shown in Table 4.2, in addition to the mean total time ($\overline{X}_{time}$), the standard deviation has been calculated to represent the spread in the subjects' times (S_{time}). The calculation shows that Korson has a small standard deviation for his modular group, yet an exceptionally large one for his monolithic

	Original		Replication	
Version	$\overline{X}_{time}$	S_{time}	$\overline{X}_{time}$	S_{time}
Modular	19.3	8.1	48.0	25.4
Monolithic	85.9	47.8	59.1	27.0

Table 4.2: Statistical values of the total times

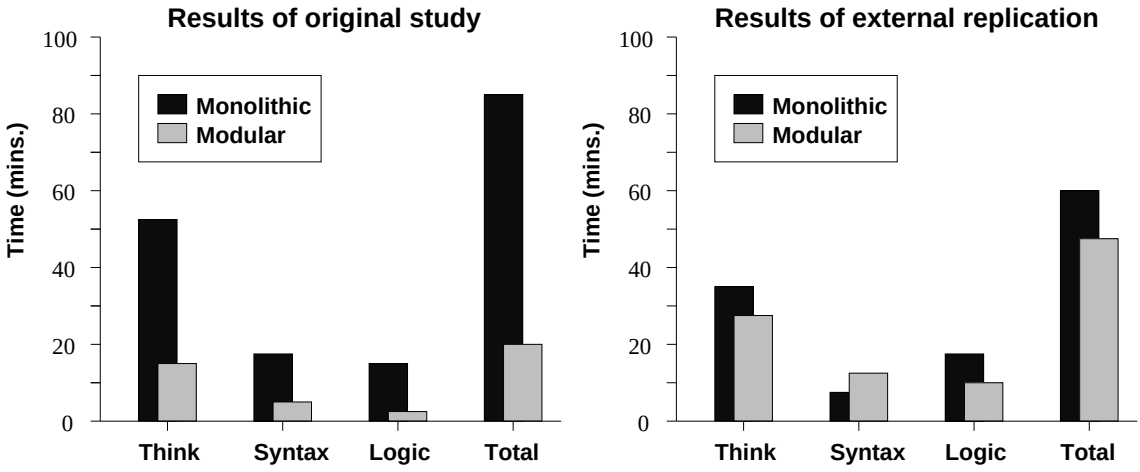


Figure 4.1: Bar charts displaying mean times for Korson and the replication

group. In contrast, the standard deviations for the replication groups were similar.

A possible reason for the relatively poor performance of the replication modular subjects could be the ability of the subjects relative to Korson's. If the subjects were less able than Korson's, however, much slower results for the modular version would also have been reflected by much slower results for the monolithic version. As can be seen from Table 4.1, this is not the case. Indeed, the opposite has occurred: in 2 out of 3 phases the monolithic times were substantially quicker than the corresponding Korson times, leading to a faster overall mean time by greater than 25 minutes. Possible reasons for this are discussed later (see Section 4.5.4).

Korson used a Wilcoxon rank sum (unrelated) test in an attempt to reject the null hypothesis, H_0 information hiding has no effect on maintainability. The probability that Korson produced his results by chance was $\rho < 0.001$ and, consequently, H_0 was rejected and Korson concluded information hiding has a positive effect on maintainability. Though the result of the replication is in the same direction as Korson's, the significance level was calculated at $\rho < 0.4$ (two-tailed unrelated t-test, $df = 15$, $t = -0.870$). This level is so far removed from Korson's significance level and even the minimal level of 0.05 usually required to reject the null hypothesis that it asks the question 'why are the results so strikingly different?', especially since Korson claimed his results were replicable. An inductive analysis was undertaken to investigate the reasons for this difference, the results of which are discussed in Section 4.5.3.

Variable	Description	Logical Values
(1) think	time to note changes to be made to program	1 .. 3
(2) edit	time to edit changes made in think	1 .. 3
(3) syntax	time to remove syntax errors	1 .. 3
(4) logic	time to remove logical errors	1 .. 3
(5) total 1	total time to complete task	1 .. 3
(6) total 2	total 1 - edit	1 .. 3
(7) program	type of program subject had	mon, mod
(8) numbchars	number of characters subject wrote in think	
(9) age	age of subject	
(10) sex	sex of subject	m, f
(11) position	university status of subject	ug, rs, ra
(12) easy	mention of task being easy	y(es), n(o)
(13) editor	mention of making good use of editor	y(es), n(o)
(14) diff	what caused the most difficulty	1 .. 3
(15) code	understanding the code	0 .. 3
(16) learn	learned anything	1 .. 4
(17) extra	any other comments	1 .. 5
(18) numbchanges	number of changes unidentified in think	1, 2
(19) StoRdiff	difference between syntax and working programs	y(es), n(o)

Table 4.3: Induction variables

4.5.2 Inductive analysis

An inductive analysis enables researchers to better understand their data through the production of facts and rules. This allows the researcher to investigate alternative interpretations of the data (as discussed in Chapter 2, Section 2.3.3). Usually inductive data has to be described at different levels of details while observing the effect on the induction rules. In the final phases of the induction process many numeric variables were assigned logical values in an attempt to induce less fragmented rules including splitting the induction data in two: one for monolithic subjects, one for modular subjects. Before discussing the rules produced by the inductive analysis package IRIS [Arisholm, 1987] (based on [Hart, 1985] and [Quinlan, 1986]), the variables used are summarised in Table 4.3. A more detailed description of variables, where required, is provided below.

Subjects' times, variables (1) to (6), were graded into three groups. These groups were determined by a) plotting a histogram of the data, b) choosing the two best split points on the histogram, and c) grading into groups. Subjects were categorised by position, variable (11), within the department in order to distinguish between undergraduates (ug), research students (rs) and research assistants (ra). Variables (12) to (17) are concerned with the answers each subject supplied to the debriefing questionnaire. These were graded into logical groups with as similar meanings as possible. Variable (14) diff was graded: 1 - pretest; 2 - finding the correct places to

	Induction Variables																		
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
A	1	1	1	1	1	1	mod	237	26	m	rs	y	y	3	3	1	1	1	n
B	1	2	1	1	1	1	mod	58	22	m	ra	y	y	1	1	2	5	1	n
C	1	1	1	1	1	1	mod	98	21	m	ug	y	y	2	2	1	1	2	n
D	1	1	1	2	1	2	mod	105	22	m	ug	n	y	3	0	3	5	1	n
E	2	2	1	1	2	2	mod	116	20	m	ug	y	y	1	3	3	1	1	n
F	2	1	2	1	2	2	mod	144	33	m	rs	y	y	2	2	3	2	2	n
G	2	1	3	1	2	2	mod	69	27	m	ra	n	n	3	1	3	3	2	n
H	3	2	3	2	3	3	mod	176	23	m	rs	n	y	3	2	3	2	2	y
I	1	2	1	1	1	1	mon	180	20	m	ug	y	y	2	0	4	2	2	n
J	2	3	1	1	2	2	mon	333	19	f	ug	y	n	3	0	4	2	1	n
K	2	2	1	1	2	2	mon	679	20	m	ug	y	y	2	2	3	2	1	n
L	3	2	1	1	2	2	mon	652	22	m	ug	y	n	2	0	1	4	1	n
M	2	3	2	1	2	2	mon	562	22	m	ug	y	n	2	1	3	4	1	y
N	2	3	3	1	3	2	mon	429	33	m	rs	y	n	2	1	1	2	2	n
O	2	3	1	2	2	2	mon	445	22	m	ug	y	y	3	1	3	2	1	y
P	3	3	1	3	3	3	mon	661	26	m	ra	n	y	3	0	1	2	1	y
Q	2	2	2	3	3	3	mon	644	33	m	ug	y	n	3	3	2	2	1	y

Table 4.4: Final data for induction package

modify; 3 - remaining mixture of individual answers. Variable (15) code was graded: 0 - no understanding; 1 - only the relevant parts; 2 - fairly well; 3 - well or very well. Variable (16) learn was graded: 1 - nothing; 2 - read the instructions fully; 3 - mixture of individual answers; 4 - to make modifications the code does not have to be understood. Variable (17) extra was graded: 1 - comments in the code not noticed; 2 - no extra comment; 3 - comments in code read but not of any help; 4 - pascal syntax forgotten; 5 - comments in code helped. Subjects' programs were saved after: (i) editing the changes made in the think phase; (ii) removing all syntax errors; (iii) the program was logically correct. Saving these programs allowed the introduction of two new variables. The number of changes unidentified in the think phase, variable (18), was divided into two groups: 1 (0 or 1 change) and 2 (2 or more changes) — calculated by comparison of the edited version to the completed, working version. Saved programs were also examined to determine if any changes were made from the syntax phase to the logic phase, variable (19) - yes or no. In addition to the experiment times, times for the pretest were also categorised as with variables (1) to (6) as in Table 4.3, e.g., pretest variable (1) was the think time for the pretest. The resulting database (shown in Table 4.4) for the subjects who completed the experiment contains the final data used by the induction package. The raw data for the replication can be found in Appendix E.

	think	syntax	logic	total		think	syntax	logic	total
Subject A	9	1	1	11	Subject I	17	3	2	22
Subject B	26	7	18	51	Subject J	16	2	0	18
Subject C	17	2	2	21	Subject K	20	1	1	22
Subject D	13	11	25	49	Subject L	17	1	7	25
Subject E	17	1	30	48	Subject M	17	1	1	19
Subject F	25	5	8	38	Subject N	25	1	13	39
Subject G	22	1	21	44	Subject O	15	1	0	16
Subject H	23	4	19	46	Subject P	14	1	18	33
					Subject Q	10	4	1	15

Table 4.5: Result times of pretest

4.5.3 Induced rules and interpretation

Rules were produced for each variable (including the pretest variables) in an attempt to explain the difference between the replication findings and Korson's. Many of the rules induced were highly fragmented, but two rules were produced which drew attention to two interesting patterns:

1. A suggested relationship was found between the total times taken for the experiment and the pretest. All 9 of the monolithic subjects appeared in the top 12 places when ranked by pretest timings (see data in Tables 4.1 and 4.5).

2. A high number of changes missed in the think phase in the experiment suggested either an excessively high syntax or logic time.

The first rule or pattern can be interpreted as an ability effect (programming ability differences of 4 to 1 to 25 to 1 have been reported [Brooks, 1980], [Curtis, 1980]). Also note that modular subject A, who finished both the fastest on the pretest and the experiment, was known to be a person with high ability. If the ability effect interpretation is accepted, an underlying problem with the replication is revealed as the majority of the higher ability participants were assigned to the monolithic task (remembering participant assignment was random and not based on pretest timings). Korson, on the other hand, claimed to have found no such ability effect with the exception of one subject who consistently finished first in his series of experiments. So the suggested ability effect is one source of variability between the two experiments.

Only subjects B, Q, and O deviate from this ability interpretation in the sense of a performance mismatch between pre-test and experiment. Comments made during debriefing helped explain two of the mismatches. Subject B, who had the longest pre-test time but the second fastest modular time, commented that the instructions were long and difficult to read, that they had not taken enough time to read pre-test instructions,

but that they had learned from this experience. This subject also mentioned that the comment preceding the four procedures where the required modifications were concentrated was a big help. Subject Q, who took the longest with the monolithic code despite being the second fastest on the pretest, commented that they had not read the instructions fully at the start and had missed important code for outputting a record. This subject had also accidentally deleted a line during the experiment and a monitor advised him of his problem. The monitor was later interviewed about this intervention. There was no doubt that the line had been accidentally deleted and it was simply coincidence that this had been observed by the monitor. When the accidental deletion was corrected the program passed the test straight away. So there is justification for including this datum in the analysis: to have discarded it would have meant discarding the longest time for a monolithic subject. Applying a correction factor to the recorded time was considered, but the monitor could only crudely estimate that several minutes had elapsed between the accidental deletion and its notification to the subject — too crude an estimate to work with. Had the subject been allowed to continue unawares, then the recorded time would have had little to do with the experimental hypothesis as the source of this difficulty was program independent. Subject O, who had a good pretest performance but a relatively poor performance on the experiment proper, commented on a difficulty having arisen in the experiment because of having read data into the wrong variable and that they hated monolithic code. This subject had also written procedures, an action which could have taken extra time.

The second rule or pattern draws attention to the fact that subjects who missed a number of changes (two or more) in the think phase had either a higher syntax or logic time. What is the significance of this? First, of the four subjects whose syntax was highest, all made edits correcting semantics during the syntax phase, specifically against instructions. Significantly, three out of the four subjects performed their tasks on modular code. These three subjects totalled 73 minutes of syntax between them and are the main reason that the mean syntax time for the modular version is greater than the mean syntax time for the monolithic version. Second, it tends to indicate that there exists a certain element of artificialness surrounding the experiment. The timing of one phase appears to be overlapping into the timing of the following one, creating potentially inaccurate timings of think, syntax, and logic. Explanations for this include: (a) the experiment itself was to blame due to poor design and the environment of think, edit, syntax and logic phases being artificial or (b) the monitors who timed the subjects (one monitor to four or five subjects) were not strict enough

in controlling when a subject was ready to move from one phase to the next.

4.5.4 Discussion

Korson required that the various versions of subjects' programs were saved as they worked through the phases of the experiment, but does not present any results or analysis from such important data. Analysis of the replication subjects' programs was conducted. The Unix utility diff allowed comparison of the saved syntax program to the saved logic program and gave an indication of what caused the subject extra time, if any, to debug. As shown in Table 4.4 from variable (19), only five subjects actually had any difference between the two programs, and this tends to indicate that subjects who didn't have any difference should have had relatively low logic times: all they appeared to be doing was running through the testing procedure. In reality this was not the case. Logic times for those in the modular group with no difference between the syntax and working programs ranges from 1 to 20 minutes, while the equivalents in the monolithic version only range from 1 to 5 minutes. Why is this? One explanation of this variability is the procedure the subjects followed to test the program. As in Korson's experiment, there were three distinct ways of carrying out a full test of the program: (i) using a single hot-key, (ii) using a series of hot-keys, and (iii) manually entering test data. The expert times for each method were: (i) 1m 15s, (ii) 55s, and (iii) 2m 30s. In addition, if the subject was to reread the instructions on the testing procedure another minute or so may have been spent. Consequently, any time between 1 and 5 minutes may be regarded as explainable. Perhaps, it may have been better practice for the monitor to test the program for the subject in attempt to reduce the significant variability that appears to have arisen. From a hypothetical viewpoint, what would happen to the results if the subjects whose saved syntax program were no different from their saved logic program were given a logic time of 1 minute, i.e., if they were given logic times assuming that they had performed the testing procedure in the minimum time possible?[1] The answer is there would be still be no significant difference between modular and monolithic subjects with $\rho < 0.259$ (two-tailed unrelated t-test, $df = 15, t = -1.172$), although the difference between the total time means does widen so that on average the monolithic subjects took 1.38 times as long as the modular subjects. To be conservative two-tailed significance levels have been used but even if one-tailed levels had been used the result would remain well outside the minimal α level of 0.05 usually required.

[1] Bringing them into line with the majority of Korson's subjects.

Another source of variability was the approach the subjects took towards making the modifications. It was noted by monitors during the experiment that several subjects attempted to follow the execution flow of the program, rather than just concentrating on finding the modifications to be made. As a consequence, subjects understanding of the code ranged from not at all to very well. In the case of subjects I and J, the two fastest finishers in the monolithic version, both subjects explained in the subsequent debriefing questionnaire that they had no understanding of the code and both subjects made the added comment that no understanding of code was necessary to modify it (this latter comment is in agreement with the feelings of some of the pilot study subjects — see Appendix E). On the other hand, examination of the listings with the written modifications showed that subjects G and H had indeed attempted to follow the flow of program control. These subjects took the longest time to finish the tasks for the modular version. Yet another modification variation was the fact that subjects L and O both wrote procedures for the monolithic version, an action which could have taken extra time.

As already mentioned the monitors may not have been strict enough compared to the monitors of Korson's experiment. This may be one of the reasons for the overlapped phases. However, a comment made by one of the subjects who didn't complete the actual experiment said the nature of the experiment made them take much longer than if they had been able to implement the modifications in the way they were accustomed to, e.g., compiling, running and experiencing the program first and then making changes. The phased approach was artificial to them. Subjects K and P made unsolicited comments to the same effect. Other participants (B and Q) commented that the instructions were too long and difficult to read resulting in additional delays.

4.6 Conclusions

The results of the replication were markedly different to those of Korson and consequently do not support his claims of replicability. Korson chose to rely on a quantitative analysis of timings. In contrast, the approach adopted for the replication was to combine traditional methods with an inductive analysis: this enabled an understanding of the behaviour of the replication subjects and yielded a number of interpretations to explain why the results were different. So by unraveling the processes several possible reasons for the variability in the subject behaviour have been revealed, thereby

braking the paradox of the experimenters' regress: the different results obtained in the replication study can be explained.

Evidence of an ability effect was found, a major source of variability between the two studies. There were several other influences on subject timings. Some subjects failed to identify all the necessary changes during the think phase. The simple act of incorrectly deleting a line took some time to be spotted by one subject. Two subjects even began to proceduralise the monolithic code. Several subjects commented on the artificiality of the experiment: one in particular wanted to engage in a prototyping approach and was unhappy at having to fall-in with the constraints of the experiment. As such, these examples illustrate the number of variables that can influence the results of an empirical study. Simultaneously, the influence of such variables demonstrate the importance of performing external replications in order to ascertain the reliability of the original results.

Chapter 5

Evaluation

5.1 Introduction

Part II of this thesis has discussed the concept of external replication and illustrated its importance to software engineering by means of the first conducted external replication study. Recently Votta and Porter have recognised this importance and have concluded that empirical studies must be repeated and the results published whether they agree or disagree with the original results [Votta and Porter, 1995]. If the software engineering community adhere to this recommendation and perform more external replications then more reliable and generalisable results will emerge over time.

Each external replication study should be categorised within the extension of the experimental framework paradigm discussed in Chapter 3. In terms of method, task, and subjects, the external replication detailed here is categorised as (`improved, similar, similar`). The method has been categorised as improved because the replication subjects were debriefed.

In other disciplines, for example, the behavioural and social sciences, researchers have debated for some time the importance of replication for establishing sound results, e.g., [Amir and Sharon, 1991], [Lamal, 1991]. From their experiences of undertaking external replications, researchers in these disciplines have produced rules and recommendations which focus on what type of replication to conduct, e.g., [Hendrick, 1991], and what should be reported to evaluate the replication, e.g., [Rosenthal, 1991]. The final chapter in this thesis part integrates the rules and recommendations of relevance for empirical software engineering with what has been learned from conducting the

external replication study presented in Chapter 4.

5.2 Lessons learned

In the following sections advice is presented to software engineering researchers regarding (i) the scale of recipe-improving when conducting an external replication, (ii) on the reporting of subject-based experiments, and (iii) general aspects of conducting subject-based experiments.

5.2.1 Scale of recipe improving

It is rare not to have ideas on how the experimental recipe can be improved upon. As stated in Chapter 4, making major improvements to Korson's experimental methods was resisted as the main interest lay in trying to perform a near exact external replication to confirm the original results, i.e., only minor recipe-improving changes were made. The significant improvement of debriefing subjects was made, but this did not interfere with the application of the original method. To highlight the problem of deciding the scale of recipe-improving, suggestions for improving Korson's experimental design are listed: improve program layout and commentary, remove biased comment in the modular program, replace the global variable in the modular program to fully implement information hiding, and reduce the size of the experimental instructions given to subjects. (One can never be sure of obtaining similar results had some or all of these improvements also been implemented).
`Recommendation`: when conducting an external replication, carefully consider its purpose. If it is to confirm the original results then only minor recipe-improvements can be made. If it is to attempt to generalise the results in some manner then major recipe-improvements or alternatives must be made. (Note well: if improvements or alternatives are too substantial, it becomes debatable whether the study counts as an external replication).

5.2.2 Level of reported detail

Once an experiment has been performed and the data analysed, the researcher must provide as much detail of the empirical research in the report as is possible to allow others to perform external replications. The volume of such material dictates that it is likely that it can only be documented in a thesis for a higher degree or with the aid of a highly detailed technical report or a laboratory kit made available on request. One

such laboratory kit for software defect-detection techniques [Lott and Rombach, 1995] has recently become available and is now being used for external replication work.

Even though Korson's work is reported in his doctoral thesis, several details were absent from the experimental reporting which introduced uncertainty into the external replication study and which disguise some possible sources of variability between the experiments. These details are presented below along with recommendations to overcome the problems that were encountered.

1. Korson did not present any information regarding the number of monitors he used: it was found that a ratio of one monitor to 4 or 5 subjects was just satisfactory. Had it been known that Korson had worked with a better monitor-subject ratio steps would have been taken to provide additional monitors.
   `Recommendation`: if making use of human monitors explicitly state the monitor-subject ratio. Better still, automate any data collection methods and use monitors only in a problem solving capacity stating clearly their experimental role and what the acceptable level of assistance given to subjects is.

2. Korson made no mention of verbal instructions issued to subjects. Was additional information given to them emphasizing particular written instructions or was nothing said? An issue is made of this because, almost without thinking, information was given verbally to students that different versions of the program existed.
   `Recommendation`: clarify any verbal instructions given to subjects.

3. With hindsight, it could have been ensured that subjects had properly completed each phase by instructing monitors accordingly. From Korson's written instructions it would seem that subjects were allowed to decide when to move on, but it may well have been that Korson's monitors applied additional control. This comment is made because there is genuine surprise that Korson did not report any problems with subjects moving prematurely between phases in the preliminary and primary experiments he conducted.
   `Recommendation`: for experiments involving subjects progressing through various phases, clarify whether subjects were required to genuinely complete phases and, if so, how this was controlled.

4. Korson provided subjects three distinct methods of testing the program for correctness, yet did not publish the likely timings for these different methods: providing variability in the method of testing may have been a source of variability

between the original and replication results.
`Recommendation`: expert timings should be reported, especially so if there is an established variation in the way subjects may perform. (The reporting of expert timings also acts as a check on the existence of ceiling effects which occur when an experimental task is too easy).

5. It is usual to address outliers in a data set, but Korson does not present any additional information to explain why one of his subjects took an exceptionally long time to complete the monolithic version: he relies solely on the interpretation bound up in the null hypothesis. Providing additional information might have benefited researchers interested in performing an external replication.
`Recommendation`: alternative explanations of data, especially outlier data, should be sought and reported.

6. Korson did not publish his pretest results. Had he done so, this would have allowed comparison with the replication subjects' times and provided another measure of the ability of these subjects relative to the original subjects.
`Recommendation`: when pretest data is collected, it should be published to assist those who are attempting an external replication.

7. The criteria Korson set for subject selection was perhaps insufficiently specific: retrospectively, it is suspected that some subjects whose experience or ability was not of the required level may have passed the selection criteria for the replication study. (Recall that six of the subjects failed to complete). While Korson qualified his professional programmer classification as someone with the equivalent of at least one year's programming experience as a full time programmer, he did not provide such additional qualification for his advanced computer science students. Any computing student who had completed a Pascal programming class could have passed the criteria set down by Korson: fluency in Pascal, knowledge of the IBM-PC, and an amount of programming experience.
`Recommendation`: more objective subject selection criteria should be stated when specifying parameters such as position, programming experience, previous experience of performing maintenance tasks, experience of software environments (languages, editors, compilers or interpreters), and experience of platform employed for the experiment. How recent that experience is may also be an important parameter. (The danger is that the criteria become too exacting and gathering enough participants for the study becomes impossible. At the same

time, the subject pool can become so specialised that external validity is compromised because of the variability amongst professional software engineers).

8. Criticism can be made of the fact that though 7 of Korson's 16 subjects were classified as being professional programmers, the distinction is not maintained in the design, the analysis, or the interpretation of results: subjects were randomly assigned. Had all the professional programmers been assigned to the modular group? It is not known.
   `Recommendation`: where recognizable differences exist between subjects, data is presented to allow an analysis taking account of these differences.

The criticisms of Korson's empirical study should not be considered severe. With the benefit of hindsight these points are easily made because the results of the replication had to be fully analysed to explain the marked difference between them and those of Korson. Korson went much further than many researchers in reporting experimental procedures and materials and he must be commended for that. He published his code for the experiments (both the pretest and the experimental code) and the instructions for both the pretest and experiment. He published individual subject timings rather than just averages, along with the statistical tests and their results. Unfortunately, a few omitted details have prevented the fullest possible interpretations of the external replication.

5.2.3 General recommendations

Finally, as a result of conducting the replication, there are several general recommendations made that researchers performing empirical work should consider.

1. In the Korson replication the information gained from the debriefing questionnaires was invaluable and provided the opportunity to understand the process individual subjects were going through. It also helped to explain the difference between the replication and the original results.
   `Recommendation:` debrief subjects after they have participated in the study by means of either a debriefing questionnaire or interview.

2. The replication results were subject to a random ability effect, i.e., the majority of the higher ability subjects were randomly allocated to one group creating two groups of uneven abilities. This was a significant source of variability between the original study and the replication.

`Recommendation:` when planning an empirical study, consider carefully the ability of the subjects and design the experiment to control for any effect ability may have on the variables being measured.

3. For the replication of the Korson experiment is was assumed that the effect being investigated was large simply because of the huge difference between the means of the two groups. Consequently, the replication was conducted with a similar number of subjects as it was felt this would provide sufficient statistical power. If it had been known that Korson thought a smaller effect was being investigated steps would have taken to recruit more subjects, thereby increasing the statistical power of the replication.
   `Recommendation:` attempt to estimate the size of the effect being investigated and the statistical power of the experiment (as discussed in Chapter 2, Section 2.3.2). This should aid researchers attempting an external replication.

5.3 Conclusions

External replication is a vital part of establishing sound results in the behavioural and social sciences. In software engineering the need for external replication has been demonstrated through a brief review of the literature concerned with performing empirical research, and by means of conducting an external replication study whose results were markedly different from those of the original. Consequently, it is argued that external replications have an important role to play in the realisation of reliable and generalisable results within software engineering. Empirical software engineering studies must be externally replicated at research centres around the world.

A set of recommendations have been introduced which researchers should consider when (i) undertaking an external replication, (ii) conducting subject-based experiments, and (iii) reporting subject-based experiments. Other researchers will wish to refine and add to this list as more external replications are performed.

An approach which reduces the problems of relying on the results of a single empirical study is the multi-method approach, i.e., using two or more different empirical techniques to investigate a phenomenon in the hope that they confirm one another. It is this approach that is introduced in Part III of the thesis.

Part III

A MULTI-METHOD APPROACH

Chapter 6

A Multi-Method Approach To Performing Empirical Research

6.1 Introduction

Depending on the aim of an empirical study, evidence may be derived from any of the following empirical techniques: measurement of industrial data, questionnaires, structured interviews, subject-based laboratory experiments, thinking-aloud protocol analysis, etc. Although each technique produces different empirical data, data collected from one empirical technique can complement data collected from another technique. This approach is termed 'multi-method'. If an effect is demonstrated by two or more different empirical techniques independently, it is more likely that the findings are reliable and will be accepted by the software engineering community. When the results agree, the empirical techniques are said to have confirmatory power.

This chapter introduces the multi-method approach for conducting empirical software engineering research and emphasizes the strengths of the approach in comparison to a single-method approach. The following chapters present an application of the multi-method approach through a three phased programme of empirical research within the object-oriented paradigm. The object-oriented approach has become increasingly popular primarily due to the alleged benefits provided by object-oriented software, yet little empirical evidence exists to support many of these claims. Section 6.3 details in full why an empirical programme of research within this area is required.

6.2 The multi-method approach

The multi-method approach is a self-developed approach for software engineering empirical research which has some parallels with the technique of triangulation and the multimethod approach in the social sciences [Brewer and Hunter, 1989], [Neale and Liebert, 1986]. In software engineering, the multi-method approach involves using two or more different empirical techniques to investigate the same phenomenon. If the results agree, the empirical techniques are said to have provided confirmatory power; more confidence can be placed in the findings. If the results do not agree, the empirical studies should not be immediately considered inconsistent:[1] although the studies may not have confirmed one another the researcher should be inspired to discover why this has happened. There may be specific reasons for the conflicting data, e.g., novice and expert programmers may have been used from one study to the next, where the benefits achieved by the experts were not demonstrated by the novice programmers. Information about when a particular software technology provides benefits and when it does not is very useful and it can contribute greatly to the cohesive body of knowledge being built from empirical study.

Applying the multi-method approach can be performed in an evolutionary manner. This approach should be adopted when the area of interest has little empirical research conducted within it and any formulated hypotheses are not particularly focused. Rather than investigating an effect through two or more different empirical techniques and hoping confirmatory power is established between them, an initial exploratory study gathering qualitative data is undertaken. At this early stage, the initial study presents the opportunity to explore a wide range of topics within the area of investigation. The collected data is then analysed by qualitative analysis techniques. Once the analysis is complete, the important findings from the initial study are refined and used as hypotheses for the next empirical study. The process is then repeated for this phase of the multi-method approach. Once the analysis of the data for the second phase is complete one of two situations can arise: (i) the results that have emerged from this phase confirms the results of the previous phase or (ii) the results do not confirm the results of the previous phase.

If there is consistency between the results of the two phases then the next phase is undertaken. This phase will investigate further the important findings of the current study and so on. This process continues until just a single hypothesis is being

[1] At least until it is certain that there is no explanation to account for the difference between the two studies results.

thoroughly investigated. For example, the evolutionary programme of research that is reported within this thesis consists of three phases. It begins with an exploratory investigation using structured interviews. The interesting findings of this initial investigation are then used to produce a questionnaire survey with the intention of confirming the findings of phase I (which actually occurs). Finally, a series of subject-based laboratory experiments are conducted which examine one of the important results from phases I and II in a more controlled setting. (Section 6.4 details the planned application of multi-method approach in full).

On the other hand, if the results are inconsistent between the two phases, investigation should be undertaken to discover why. If no explanation can be provided then the researcher is faced with a dilemma: (a) should the important results of the latest study be refined and investigated further by another study or (b) should the empirical study be redesigned and repeated in order to discover which phase has produced more reliable results. This dilemma may not be as serious if more than two phases of the programme of research have been conducted — if it is only the last phase that does not confirm the findings, it is probable that its results have been influenced by a uncontrolled variable and, hence, are not reliable.

6.2.1 Strengths and weaknesses

In addition to providing confirmatory power, a multi-method approach has several strengths that a single-method approach does not have.

Hypothesis formulation. Hypothesis formulation is no longer solely dependent on the researcher's intuition: by conducting an exploratory study, a wide range of important topics within the area of interest can be investigated. The results of this study are then used to formulate several initial hypotheses for further investigation through another technique. The process is repeated until a single hypothesis is being investigated, i.e., each phase of the programme of research becomes increasingly focused on the important issues — other less important issues can be investigated later. An intangible benefit of this is that when conducting laboratory experiments during later phases of the programme of research, estimation of the effect size becomes easier because the researcher has data upon which to base their estimate. As a result, statistical power analysis can be more readily performed (see Section 2.3.2).

Robustness of conclusions. The programme of research consists of a number of phases each conducted using, e.g., a different number of subjects from a different sample of the population, by means of a different empirical technique. If results are consistently demonstrated across each phase, conclusions drawn at the end of the programme of research are less likely to be adversely affected by the fact that one of the phases was poorly designed, used a biased sample of the population, misapplied statistical tests, or did not take into account one of a number of other important factors.

Increased understanding. The multi-method approach can result in deeper explication of the important factors affecting the phenomenon under investigation. As each phase of the programme of research becomes more focused, the researcher is likely to gain increased understanding of the various factors that affect the phenomenon.

As a consequence of these strengths, results emerging from a multi-method programme of research are more impressive than those from a single empirical study. The software engineering community are, therefore, likely to accept the results as more reliable and generalisable. Further, independent researchers are more likely to try and replicate the important findings of the programme of research (the advantages of which are discussed in Part II of this thesis).

The main weakness of the multi-method approach is the initial investment of time and effort required to apply it: each phase must be planned, designed, analysed, and fully reported before the next phase can begin. Hence, a large investment is required for a three phased programme of research and little return on that investment will be achieved until after the initial phase(s). For example, in the initial structured interview phase detailed in Chapter 7, it was found that each of hour of recorded material required approximately 25 hours of effort to transcribe, summarise, and fully analyse. On the other hand, it is argued that this is a worthwhile investment because of the strengths the approach provides. Overall, a focused programme of research is likely to be more cost effective than several independent studies because more reliable and generalisable conclusions can be achieved.

6.3 Why the object-oriented paradigm?

The object-oriented paradigm seems set to dominate the software engineering industry. This appears to be largely as a result of market forces and the opinion of gurus.

By incorporating the features of data abstraction, dynamic binding, encapsulation, inheritance, and polymorphism, object-oriented software development has, according to its advocates, many advantages over conventional methods. For example:

- Object-oriented design is more suited to real world problems [Booch, 1986], [Rumbaugh *et al.*, 1991], [Schlaer and Mellor, 1988].
- Object-oriented software is easier to understand [Booch, 1986], [Pokkunuri, 1989].
- Object-oriented software better facilitates maintenance and enhancement [Berard, 1993], [Booch, 1986], [Rumbaugh *et al.*, 1991].
- Object-oriented software resists degradation [Booch, 1986], [Pokkunuri, 1989].
- It is easier to reuse object-oriented software components [Berard, 1993], [Booch, 1986], [Liberherr and Xiao, 1993].
- Programmer productivity is improved [Buzzard and Mudge, 1985], [Booch, 1986].
- The total life cycle costs can be reduced [Buzzard and Mudge, 1985].
- Testing of object-oriented software is easier [Rumbaugh *et al.*, 1991].

Unfortunately, there currently exists little empirical evidence to support such claims. For example, Jones details a visible lack of empirical data to support the assertions of substantial gains in software productivity and quality, reduction in defect potential and improving defect removal efficiency, and reuse of software components [Jones, 1994]. Henry *et al.* provide a list of references which they state have made claims having qualitative appeal, but which have little supporting quantitative data [Henry *et al.*, 1990]. Evidence in support of the object-oriented paradigm is slowly beginning to filter through in certain areas, but is by no means conclusive. For example, Mancl and Havanas provide some supporting evidence for increased software reuse and ease of maintenance through better data encapsulation, although their study may lack the reported detail to convince the strongest skeptics [Mancl and Havanas, 1990]. Deubler and Koestler extended and re-implemented subsystems of the operating system BS2000 using the object-oriented paradigm and embedded them within the existing code written in a combination of low-level and high level languages [Deubler and Koestler, 1994]. The authors' experiences lead them to believe the object-oriented code produced: (i) was more structured and easier to understand, (ii) had substantially less errors than

was normal for a new piece of software, and (iii) had less interdependencies among software components.

But it is not all good news. Deubler and Koestler also stated that the object-oriented code produced was 10 - 15% less efficient than the original implementation. More importantly, a subsection of the literature has recently reported that the same object-oriented concepts which lead to a proliferation of positive claims about the paradigm may actually cause problems for programmers. Alleged difficulties exist in understanding, maintaining, testing, and reusing object-oriented software and in the capacity of class hierarchies to withstand change. These are described in more detail in the following sections.

6.3.1 Understanding object-oriented software

Object-oriented design localises data in objects to hide data structures, but potentially distributes a program function across several or more classes. Distribution of functionality, i.e., where pieces of code that are conceptually related are physically located in non-contiguous parts of the program, is termed a delocalised plan [Soloway *et al.*, 1988]. In object-oriented software the mechanisms of inheritance, polymorphism, and dynamic binding are responsible for the creation of delocalised plans [Wilde and Huitt, 1992]. Furthermore, much of the work on good object-oriented programming style encourages the use of small methods, e.g., [Liberherr and Holland, 1989], and this exacerbates delocalisation. 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 [Kung *et al.*, 1994], [Lejter *et al.*, 1992]. For example, understanding the effect of a particular method may require tracing a chain of method invocations to find where the work is being performed.

It is also reported that when interacting with a software system for the first time, a programmer requires a method to identify the components and perceive the system architecture. In conventional systems, this can be achieved by examining the module call hierarchy beginning at module main. In object-oriented systems this call hierarchy will be a hierarchy of methods which has several disadvantages: (i) dynamic binding may make this hierarchy difficult to calculate, (ii) there may be no 'main' method in the system, and (iii) a hierarchy of methods detracts from the grouping of methods within objects. Consequently, it is claimed that it can be difficult to gain a high level understanding of an object-oriented system [Wilde and Huitt, 1992].

Ponder and Bush suggest that polymorphism can also have a detrimental effect on program understanding [Ponder and Bush, 1994]. They argue that polymorphism is useful, but creates understanding difficulties because the operations performed can only be determined dynamically. The authors explain that although it is not necessary to know the implementation of a method to understand its function, the following difficulties can arise: (i) with just a few instances of use, inferring the generic meaning of an operation is difficult and (ii) if a name is shared by subtly different operations, the programmer is encouraged to infer similarities that do not exist.

6.3.2 Maintenance of object-oriented software

Successful maintenance requires two attributes: the ability to make changes easily and an in-depth understanding of the software structure and behaviour. It therefore follows that the difficulties in understanding object-oriented software have implications for its maintenance. For example, delocalised plans must be fully understood; naming confusions caused by polymorphism must be resolved, otherwise subtle errors may be introduced during maintenance; tracing method invocations to their source must be performed, a difficult and time consuming task exacerbated by dynamic binding.

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 [Kung *et al.*, 1994], [Wilde and Huitt, 1992]. If a dependency exists between two entities in a system, e.g., Y depends on X, then a programmer modifying X must be concerned with possible side effects in Y. The number of dependencies that must be considered in an object-oriented system is far greater than in a conventional software system and, as a consequence, a maintainer can have great difficulty identifying the impact of their changes [Kung *et al.*, 1994].

6.3.3 Testing of object-oriented software

Jüttner *et al.* discuss testing of object-oriented software and in particular integration testing [Jüttner *et al.*, 1994]. Integration testing is defined as testing the interaction of program parts during the process of integration, that is the putting together the parts of a program to form bigger ones. In object-oriented software, these parts are classes. The features of object-oriented software make integration testing quite different from that of conventional software. In object-oriented programming, class and integration testing are not as clearly separated as module and integration testing in conventional

programming, primarily due to the large number of dependencies created by inheritance and polymorphism. Further, class encapsulation hides information, such as state variables and other objects, which make it more likely that faults remain hidden during the test process. The authors conclude that integration testing is more complex for object-oriented software and more research is required within the area to produce adapted and completely new integration testing strategies.

As mentioned above, the large number of dependencies in an object-oriented system make it difficult to anticipate and identify the "ripple effect" of changes made by maintenance. It is therefore difficult to identify which parts of the system need to be retested [Kung *et al.*, 1994]. In addition, the authors state that data, control, and state behaviour dependencies make it difficult to prepare and generate test data to retest the affected components (a procedure known as regression testing).

6.3.4 Reuse of object-oriented software

For reuse to be carried out effectively it is necessary to be able to locate the relevant pieces of code as quickly as possible. It is now being suggested that this is not as easy a task in object-oriented software as once thought — the large number of similar classes, each of which tend to encapsulate a large number of small methods, make it difficult to quickly browse them and find the functionality required for reuse. For example, Wilde and Huitt state that it can be difficult to find the right class to use out of Smalltalk's many different classes of Collection [Wilde and Huitt, 1992]. If the reuse benefits of the object-oriented approach are to be realised, it must be possible to locate the required code efficiently.

6.3.5 Conceptual entropy of class hierarchies

All systems that are frequently changed tend towards disorder, a characteristic recognised as entropy. Given frequent change, class hierarchies in object-oriented systems can be expected to exhibit entropic tendencies. This has been termed conceptual entropy by [Dvorak, 1994]. Conceptual entropy is manifested by increasing conceptual inconsistency as the hierarchy is traversed. That is, the deeper the hierarchy the less likely that a subclass will consistently extend or specialise the concept of its superclass.

According to Dvorak, conceptual entropy occurs for three reasons: (i) concepts become more complex — the number of properties that a developer must consider to understand a class concept increases with each level, (ii) there are more classes to consider as superclasses, and (iii) classes become more specialised, thus conceptual matches for

Object-oriented concept	Reference
Aggregation	[Jüttner *et al.*, 1994]; [Kung *et al.*, 1994]
Dynamic binding	[Jüttner *et al.*, 1994]; [Kung *et al.*, 1994]; [Lejter *et al.*, 1992]; [Wilde and Huitt, 1992]; [Wilde *et al.*, 1993]
Encapsulation / Class concept	[Jüttner *et al.*, 1994]; [Kung *et al.*, 1994]; [Wilde *et al.*, 1993]
Inheritance	[Crocker and von Mayrhauser, 1993]; [Dvorak, 1994]; [Jüttner *et al.*, 1994]; [Kung *et al.*, 1994]; [Lejter *et al.*, 1992]; [Wilde and Huitt, 1992]; [Wilde *et al.*, 1993]
Polymorphism	[Crocker and von Mayrhauser, 1993]; [Jüttner *et al.*, 1994]; [Kung *et al.*, 1994]; [Ponder and Bush, 1994]; [Wilde and Huitt, 1992]; [Wilde *et al.*, 1993]
Small methods	[Jüttner *et al.*, 1994]; [Ponder and Bush, 1994]; [Wilde and Huitt, 1992]; [Wilde *et al.*, 1993]

Table 6.1: Object-oriented concepts reported to cause difficulties to programmers

subclassing must be more exact. The concept was empirically demonstrated through an experiment using seven subjects each with on average one years object-oriented experience. Dvorak concluded that if conceptual entropy was left unchecked it would eventually reach a stage where the utility of the class hierarchy becomes constrained and it has to be restructured.

6.3.6 Summary

To summarise, the purported advantages of the object-oriented paradigm are the driving force behind its increasing popularity. The majority of these advantages, while widely publicised in software engineering text books, have little or no supporting empirical evidence. Recently, however, the literature has begun to report that the object-oriented concepts of aggregation, dynamic binding, encapsulation, inheritance, polymorphism, and small methods can cause programmers difficulties when working with object-oriented software. Again, little empirical evidence exists to support how significant these difficulties are, but because they contradict the opinion of object-oriented advocates they have been discussed in more detail. (Table 6.1 comprehensively summarises the literature which reports on any individual concept). It is therefore argued that conducting empirical work to investigate the varying opinions within the object-oriented literature is important, and timely, research.

6.4 The planned multi-method application

Clearly the object-oriented paradigm requires more empirical evidence than currently exists. As a consequence, a multi-method approach was undertaken to investigate the alleged advantages and disadvantages of the object-oriented approach. This programme of research is regarded to be evolutionary: the findings of one phase were the basis of the focus and design of the next phase. The strategy to conduct this programme of research was carefully considered to include the most useful and worthwhile techniques (as well as the most economically viable due to financial limitations). Letters sent to managers of industrial institutions throughout Scotland secured the availability of seven employees of a large corporate company who were all experienced object-oriented developers. To elicit as much as their knowledge as possible the most appropriate technique to use was structured interviews. The first phase of the evolutionary programme of research was to conduct an exploratory investigation by means of structured interviews with these industrialists and academics with industrial experience. The full details of this study are presented in Chapter 7. The findings of this primary investigation were used to design and implement a questionnaire on key aspects of object-oriented systems, which was then distributed to appropriate electronic newsgroups and to members of an object-oriented postal mailing list. The intention was to confirm (or otherwise) the findings of the first phase across a larger and wider practitioner group. The details of this study are presented in Chapter 8. Finally, a series of subject-based laboratory experiments were conducted which tested one of the important and most interesting results from the questionnaire survey and structured interviews in a more controlled setting. Full details are described in Chapter 9.

Chapter 7

Phase I: Structured Interviews

7.1 Introduction

This chapter presents the first phase of the multi-method programme of research, an exploratory investigation to elicit knowledge from experienced object-oriented developers by means of structured interviews. Structured interviewing is an appropriate method to *initiate* evidence gathering for several reasons: (i) interviewing elicits more of the subject's general knowledge than other techniques, (ii) interviewing software developers from both academia and industry should determine if academics are exploring issues that industrialists deem important, i.e., do academics and industrialists have similar or differing opinions on object-oriented issues?, (iii) interviewing experienced object-oriented software developers from industry provides external validity in the form of real world data, something which can be difficult to provide in more controlled experimentation, (iv) object-oriented software developers have experience-based opinions on the advantages and disadvantages of the paradigm; these opinions could be used to justify if an empirically unsupported practice warrants further, more detailed investigation, and (v) the information obtained from the interviews can identify issues that have not been previously considered. Section 7.2 discusses the advantages and disadvantages of using structured interviews as an empirical technique.

The structured interview requires a template from which the interview can be directed. The subject may give an answer which is deemed worthy of pursuing, and the interviewer can then ask supplementary questions (though care must be taken not to bias the interview direction). Once satisfied, the interviewer returns to the next

question on the template. Section 7.3 details in full the interview method used for this study. The remainder of the chapter provides analysis and discussion of the verbal data collected (Section 7.4).

7.2 Interviewing as an empirical technique

Interviewing is an appropriate means for conducting a primary investigation and offers several advantages over other empirical techniques:

External validity. Interviewing experienced subjects allows gathering of real world data which is of extreme importance for drawing generalisable conclusions (assuming large amounts of subjects are interviewed). Furthermore, if a number of independent experienced developers hold the same opinion on a particular issue, the likelihood is an effect of some kind exists; further empirical investigation is justified.

Flexibility. Interviewing is a flexible technique that does not have the rigidity of, e.g., a questionnaire survey, yet still allows data to be gathered relatively quickly and unhindered [Sinclair, 1990]. Interesting points made by the subject can be explored further by efficient question probing. In addition, feedback from initial interviews can be used to refine the interview template to concentrate on issues not previously considered. Conversely, refinement can remove issues which were initially considered to be important, but which, on discussion, appear less significant, i.e., the interview template can be re-oriented if necessary without major disruption to the study, as long as the detail is reported in the findings.

Motivation of subjects. The interviewer has the ability to motivate the subject to contribute additional information about a topic. The interviewer can also direct and accelerate the flow of the interview thereby improving the quality of the subject's information flow [Biemer *et al.*, 1991].

Clarification through re-interviewing. The ability to re-interview subjects allows further investigation of important topics only discussed briefly in the original interview. This elicits further detail as well as clarification of any opinions expressed, an important advantage that other empirical techniques, e.g., questionnaires, do not readily allow [Biemer *et al.*, 1991].

As important, however, is to realise that there can exist difficulties in collecting data from structured interviews:

Subject pressures. There is the danger of subjects saying what they think is appropriate to say. For example, if the subject being interviewed thinks that the listener is more proficient or knowledgeable in the interview domain then there may be pressures to appear rational, knowledgeable and correct. Inversely the subject may become uncooperative or present a particular attitude to the topic [Bainbridge, 1990]. Social influences may also affect the course of the interview, e.g., if the subject thinks what they say can be held against them then this may bias their answer towards what they think the interviewer wishes to hear. Another problem is that while an experienced subject may be opinionated and talk freely on any topic, they may not mention what they think is obvious.

Cognitive issues. Most people think more quickly than they talk (especially if time pressures exist on the subject, e.g., the subject is being asked to speak their thoughts while performing a task of some description). This highlights the general point that verbally communicated data may only give a limited sample of the total knowledge of the subject under study, i.e., only a subset of their cognitive activity may be reported [Bainbridge, 1990].

Interviewer experience. The interviewer requires a good working knowledge of the domain being questioned: without it, important aspects of the interview may be found to be ambiguous, unresolved or, simply, unnoticed.

In addition, there are no objective independent techniques for doing analyses on interview data. This may lead to subsequent problems for the analyser who may have to make assumptions when interpreting what the subject has mentioned, or use their own knowledge of the domain to make particular inferences. There are also high costs involved in terms of time and effort to fully analyse verbal data. For example, Marchioni states the need to spend about 25 hours for each hour of subject observation on tasks such as transcribing video tapes and coding detailed protocol analysis [Marchionini, 1990]. Neilson denies this and reckons in realistic development situations each hour of thinking-aloud observation only needs half an hour of combined analysis and report writing [Neilson, 1992]. In the author's experience of transcribing and analysing structured interview data, Marchioni's estimate was found to be reasonably accurate.

To summarise, the findings of an interview study are unlikely to be generalisable unless large numbers of subjects are interviewed and measurement errors are factored into the analysis [Biemer *et al.*, 1991]. The structured interview technique is very useful, however

- For gaining insight into the knowledge of others.
- For gaining insight into subjects' opinions about published problems and advocates' claims.
- As initial evidence for justifying further empirical investigation.

Papers in the literature have used similar techniques, such as thinking-aloud protocols and knowledge acquisition through video recording, apparently with success, e.g., see [Denning *et al.*, 1990], [Walz *et al.*, 1993]. It is argued, however, that while interviewing may make the preliminary survey it should be followed by the chain and transit of objective measurement. The multi-method approach is an appropriate method of achieving this. (For a more comprehensive discussion of verbal data see [Biemer *et al.*, 1991], [Ericsson and Simon, 1984], and [Goguen and Linde, 1993]).

7.3 The interview method

The interview questions were designed to elicit information on three aspects of developers' object-oriented knowledge: (i) information on their background, e.g., experience, language familiarity, (ii) their opinions on the perceived advantages and disadvantages of the paradigm, and (iii) their opinions on existing object-oriented technology, e.g., software tools, languages such as C++, and class libraries. The interview was structured in such a way that factual questions (requiring just one or two sentence answers) were asked first, followed by questions requiring answers based on opinions from experience. This approach was taken to allow the subject time to settle and relax before asking the more thought provoking questions.

A draft interview template was written and piloted during an initial interview with an experienced industrial employee. The goal of the pilot was to receive reactions on the structured interview technique and the questions asked. The only point made by the subject was that not enough questions were asked about software tools, stressing that tools are an important part of object-oriented programming. The draft template was revised to take account of this noted point and the other interviews were conducted with this refined template as their basis. (Appendix B contains a copy of this template).

Twelve subjects, chosen for their experience of working with object-oriented systems, agreed to be interviewed using the refined template. Seven of these subjects were industrial employees and five were academics, four of whom had industrial experience

of developing or working with object-oriented systems. The subjects were given no prior knowledge of the content of the interview except that it was based on object-oriented systems, but were told that any replies given would be treated confidentially to reduce concern about how their answers might be used. The interview process was kept as informal as possible, e.g., the subjects were drinking coffee before and during the interview to help them relax and answer questions freely.

The interviews were carried out during the working day and lasted from 30 minutes to almost 2 hours. Each interview was recorded using a hand held recorder which was then transcribed to paper to ease analysis. Note that the initial interview was also transcribed and included in the analysis.

It was hoped that the interview template was comprehensive, but as a safeguard each subject was asked the final question

> Has there been anything that you have not been asked that you feel is important and should be addressed?

The majority of subjects answered no to this, or mentioned something specific about their own work which they would have enjoyed talking about. In all, the subjects appeared to think the interview was a "pretty full questionnaire".

7.4 Analysis and discussion

Analysis was undertaken by summarising each interview transcript, and tabulating answers (paraphrased) for each question to compare and contrast them. This method of analysis has the advantage of allowing the data to be easily visualised, bearing in mind that 13 subjects were interviewed.

Table 7.1 summarises the attributes of the interviewed subjects: column one provides a subject identifier, column two (Pos.) represents the subject's position as either an academic (A) or industrialist (I), column three (Exp.) gives the experience of the subject in years, and the remaining columns present the subject's knowledge of various object-oriented languages. The level of object-oriented experience varied from six months to ten years, but note that the least experienced used object-oriented techniques every working day. So it is argued there is justification for describing all interviewed subjects as experienced object-oriented developers (only I, J, and L were not daily users).

			Object-Oriented Languages						
	Pos.	Exp.	C++	Flavours	Eiffel	Objective-C	OO-Pascal	Simula	Smalltalk
A	I	10	√	√	√				√
B	I	0.5	√						
C	I	3	√	√					
D	I	4	√	√			√		
E	A	2	√						
F	I	1	√						
G	I	3.5	√					√	
H	I	5	√			√			
I	A	5	√				√		√
J	A	3	√			√	√		
K	I	1.5	√			√			
L	A	2	√			√			
M	A	3	√		√	√			

Table 7.1: Subjects' position, experience, and object-oriented language familiarity

7.4.1 Learning curve, documentation, time pressures and quick fixes

Learning curve. Several subjects (A, I, J, and L) mentioned the learning curve as an influencing factor when changing from the structured design paradigm to the object-oriented paradigm: the subjects made the point that the learning curve is a steep one. In addition, subject J commented "you really have to think differently ... I'm still thinking in structured C in some ways" (repeating the point made in [Wilde *et al.*, 1993]). Subject L quantified by stating that to fully make the transition can take as long as two years.

One reason that can make the learning curve steeper than necessary is the use of hybrid programming languages like C++. Subjects B, K, L, and M said the use of C++ has a detrimental influence on learning how to develop object-oriented software. As subject K stated, "it's a language which doesn't really encourage to think in terms of objects." Subject B added, "someone ... might just regress into doing something in a C fashion when they should be doing it in an object-oriented fashion." And subject M hypothesised, "being forced to be object-oriented I'm sure actually helped me." Lozinski details many of the limitations of C++ and provides some excellent accompanying discussion by comparing it to the more highly regarded hybrid language Objective-C [Lozinski, 1991]. (Tables B.11, B.22, and B.24 in Appendix B provide individual comments).

Design documentation. Ten subjects, including all industrial subjects, indicated that they had had trouble with the availability of design documentation: either it did not exist or it was inadequate. Subject K reasoned that when time constraints become

a factor, documentation is the first thing that suffers (subjects A and D made similar statements). However, subject L warned that documentation for object-oriented systems is a more important aid to understanding than it is for other systems and if it was missing maintainers would suffer severe consequences. Subject G qualified, "design documentation is important otherwise you are back to where you were before [with structured designed systems]." The subject made the point that object-oriented systems are not easier to understand than other systems unless the design documentation fully captures class relationships, class member functions, member variables, and so on. Subject M supported this claiming, "the documentation is essential." Subject H provided the rather worrying comment, "... on the last project I worked on there wasn't any decent documentation anyway, ... you might find that, that's by far the most common case that there's no good up to date design documentation." These difficulties appear indicative of what Kung *et al.* have experienced. They state,

> Our experience indicates that it is extremely time consuming and tedious to test and maintain an OO software system. This becomes even more acute when documentation is either missing or inadequate [Kung *et al.*, 1994].

(Tables B.18 and B.25 in Appendix B provide individual comments).

Time pressures and quick fixes. Eleven subjects thought it was possible to perform quick fixes on object-oriented code and especially C++ code, e.g., "yes, particularly in C++", "it's too easy in C++", "in C++ it's a lot easier", "it's all too easy, it's really easy, and the temptation is always there". Subject L reckoned this is "one of the dangers in these hybrid languages" while subject G stated, "yes, but that's bad maintenance rather than bad design." Subjects A, C, D, F, and K mentioned a quick fix can be the result of time pressures. Analysis of the data from the external replication detailed in Chapter 4 discovered that subjects performed pragmatic maintenance (performing the minimal amount possible to complete the required task) under laboratory conditions. (Tables B.24 and B.25 in Appendix B provide individual comments).

7.4.2 Inheritance and high level understanding

Design. Subjects A, C, F, G, I, and L made the point that if inheritance is designed properly (the hierarchy is structured using appropriate abstractions) then it will not cause understanding difficulties. Additionally, 5 of these subjects agreed that if designed properly, inheritance should aid understanding. On the other hand, subjects

D, E, H, J, K, and M mentioned that inheritance can make understanding difficult: subject D said that it can be difficult to trace the flow of control, while subjects J, K, and M mentioned that tracing a line of method invocations to determine which method is performing the work is difficult (repeating points from [Lejter *et al.*, 1992] and [Wilde and Huitt, 1992]). Subjects H and K agreed it can be difficult to understand what functionality a line of code is performing until it is realised it calls an inherited member function. Subjects H and K also agreed inheritance can cause confusion when interacting with inherited member variables — where have they been inherited from? (repeating points in [Lejter *et al.*, 1992]). The statement by subject C, made in the context of design, appears to best fit the general opinion

> "I think on the whole, it comes back slightly to how well written the code is ... I think once you get into the way of thinking about inheritance it does not make it more difficult to understand and in fact often makes it a lot simpler; if your code is badly written then it's a nightmare."

(Table B.4 in Appendix B provides individual comments).

Delocalised plans. In delocalised plans, conceptually related pieces of code are physically located in non-contiguous parts of a program [Soloway *et al.*, 1988]. Inheritance creates further opportunities for delocalisation by spreading method functionality over the hierarchy. Subjects were asked about the effect this had on system understanding. Almost every subject commented that if the inheritance hierarchy is designed properly then it would not be detrimental to understanding. Additionally, subject A said that it was a natural way of writing and understanding code, subject C declared that "it's likely to aid understanding", and subject H stated "it simplifies things; it simplifies your design". (Table B.5 in Appendix B provides individual comments).

Depth of inheritance. Subjects were asked to define a deep inheritance hierarchy and the effect of depth of inheritance on system understanding. These definitions are summarised in Table 7.2. Subject C said that 10 levels of inheritance (including multiple inheritance) would be a deep and complex hierarchy. The subject added,

> "you want to avoid using a depth of nesting that you don't need and it's therefore inappropriate."

The subject made the point that the hierarchy must be appropriately designed. Subject H also noted that 10 levels of inheritance is a deep hierarchy. In contrast, subject

	Depth of inheritance (levels)			
Subject	3-4	5-6	> 6	Not relevant
A	√			
B	√			
C			√+ MI	
D		√		
E	√			
F	√			
G	√			
H			√	
I				√
J	√			
K		√+ MI		
L				
M		√		

Table 7.2: Subject opinions on a deep inheritance hierarchy

I stated that depth was not an issue they had ever considered and, therefore, the question was not relevant. The remaining subjects mainly noted 3-4 levels of inheritance as deep. Comparison of the data in Tables 7.1 and 7.2 shows no significant relationship between experience or the language used[1] and the depth of inheritance.

Multiple inheritance. Subjects B, C, G, H, I, and K mentioned that multiple inheritance is more a complex concept than single inheritance. Additionally, subject L was concerned that multiple inheritance allowed for a sloppy approach (inappropriate inheritance). The subject stated their reluctance to use multiple inheritance by mentioning anything designed using multiple inheritance can be designed using single inheritance. Subjects B and H agreed, mentioning that multiple inheritance makes your design more complex. On the other hand, subjects C, E, and K stated multiple inheritance had benefits. Subject C reckoned it is easier to reuse with multiple inheritance and it can also make maintenance changes easier. Subject E mentioned it maps the reality of systems being modeled, and subject K agreed reporting that if you have a real case for multiple inheritance then it is advantageous. Subjects B, G, and L disagreed. Both B and L stated multiple inheritance is more tightly coupled than single, and therefore software reuse can be more difficult, while subject G stated "single inheritance is the way to do it." (Table B.9 in Appendix B provides individual comments).

To summarise, Table 7.3 provides the advantages and disadvantages of inheritance mentioned by the interview subjects. (Tables B.7 and B.8 in Appendix B provide individual comments).

[1] Although most subjects talked in the context of C++ for the majority of the interview.

Inheritance	
Advantages	Disadvantages
Allows design at the highest level of abstraction	Understanding is difficult if not well designed
Prevents code redundancy	Overuse or inappropriate use of inheritance leads to more complex code that's harder to follow
Aid to understanding if designed well	Multiple inheritance can complicate the design
Information hiding through encapsulation	Tracing flow of control and dependencies can be difficult
Quick prototyping	New concept to learn
Single inheritance encourages good design	Deep hierarchies can cause understanding difficulties
Multiple inheritance for complex designs	
Provides modularity	

Table 7.3: Summary of the advantages and disadvantages of inheritance

7.4.3 Maintenance of object-oriented programs

Facilitating change. Subjects A, B, C, D, E, F, G, H, K, and L stressed that if well designed changes are made to a well designed software system, the effects of these changes are likely to be more localised with object-oriented code than with equivalent structured code. In contrast, if the changes are not well designed, then side effects will have as much an impact on an object-oriented system as on any other system. Although subjects D, H, J, and K agreed that changes are more localised in an object-oriented system, these four subjects also noted that changes can also have a greater global effect because any changes made will affect subclasses which inherit them (repeating the point in [Jüttner *et al.*, 1994], [Kung *et al.*, 1994], and [Wilde and Huitt, 1992]). On the other hand, subjects D, H, and K noted that this concept has the benefit of being able to "fix it for one, fix it for all": if class X has a corrective maintenance change made then any subclass Y has that change made automatically through inheritance (a benefit not mentioned in [Kung *et al.*, 1994] or [Wilde *et al.*, 1993]). Subjects B, D, G, J, and L raised the issue of tracing of method dependencies through the class hierarchy as "problematic" (repeating the points in [Wilde and Huitt, 1992] and [Lejter *et al.*, 1992]). According to subject K "just seeing where a piece of code does that work ... from the point of view of understandability can be quite tricky." The subjects agreed this is compounded when the inheritance hierarchy is deep. (Tables B.2, B.4 and B.6 in Appendix B provide individual comments).

Object-oriented software difficulties. Regarding object-oriented programs causing problems for software maintainers, subject A related it to the learning curve: understanding the way an object-oriented program works is not easy to begin with and takes time to acquire the correct method of thinking, "learning by osmosis" (repeating the point made in [Wilde *et al.*, 1993]). Subject J repeated that missing documentation

could cause maintainers difficulty. Subjects B and K mentioned the one class per file style promoted by Stroustrup [Stroustrup, 1991] as undesirable: it can make tracing of dependencies extremely difficult without good tool support (repeating the point in [Lejter *et al.*, 1992]). The general opinion, however, was that there is more structure to a well designed object-oriented system and, as such, it is easier to maintain than a non object-oriented system. As a caveat, subject L stated that perhaps too few object-oriented systems have had sufficient maintenance to be able to say. (Tables B.10, B.11, and B.16 in Appendix B provide individual comments).

Rate of entropy. Subjects were asked if they thought that continual maintenance would eventually lead to unmaintainability. The majority of subjects commented that this would be the case although in comparison to an equivalent structured program there is a lesser tendency for this to happen, i.e., it would still happen but over a longer of period time. Subject A stated this would happen "without a doubt": the rate of entropy increases over time and everything tends to disorder. Subject D also discussed the rate of entropy but did think it is reduced for object-oriented systems. Subject E agreed, but only on the condition that programmers "maintain the object mindset for making changes" to the system. Subject J made a similar statement. Subject H concluded that as with anything, if ad hoc changes are made then unmaintainability will ensue. (Table B.12 in Appendix B provides individual comments).

Software tools. Subjects mentioned that tools are helpful for understanding and maintaining object-oriented programs. Differences of opinion did arise about whether they are a necessity or not: subjects E, F, H, J, K, L, and M all declaring they are, the remainder declaring they are not. All the subjects did agree that tools do alleviate some of the understanding problems that can occur when performing object-oriented software maintenance. Subject I reckoned that more object-oriented tools are needed. (Table B.3 in Appendix B provides individual comments).

Quality of maintenance. Subjects A, B, C, D, G, K, and M mentioned that it would be difficult for a maintainer to tell if any changes made had degraded the system code quality (repeating the point made in [Kung *et al.*, 1994]). Code reviews and walk-throughs performed by a panel of programmers are recommended practice by these subjects as a reliable safety net for catching any introduced errors. (Table B.17 in Appendix B provides individual comments).

7.4.4 Other issues

Small methods. Regarding small methods as good object-oriented programming practice, subject A concluded that methods "can be one line ... but as much as 50 lines for 'real' methods" depending on their function. Subject F stated, "I try to keep them small." However, subjects B, C, D, and K all mentioned methods of size in excess of 100 lines of code (subject C had worked with a method of 2500 lines) although this was not regarded as good programming practice. In general, the subjects felt that small methods are good programming practice. (Table B.1 in Appendix B provides individual comments).

Object-oriented programming practices. Subjects were asked if they had ideas on good and bad object-oriented programming practices. Once more the importance of design was stressed. Subjects A, C, F, I, K, and M mentioned that good design usually means good code; conversely bad design produces bad code. According to subject C "it is almost less the coding style than the design style that is important". Additionally, subjects C and D agreed that not too deep an inheritance tree was good style (although from Table 7.2 it can be seen their definitions of deep differ). Subject H added "there seems to be few points of reference for object-oriented code quality". The rules and recommendations produced by Henricson and Nyquist for object-oriented programming is one standard which may help to reduce any existing inconsistencies [Henricson and Nyquist, 1992]. (Table B.16 in Appendix B provides individual comments).

Polymorphism. Every subject interviewed thought that polymorphism is a useful concept in object-oriented programming and they agreed it can produce generic code. However, subjects B, C, E, G, I, and J mentioned that polymorphism, especially if semantic consistency is not maintained, can cause confusion and make it difficult to understand how the system works. In particular, subject B stated,

> "Yes, it's a great benefit in some cases ... In some cases, it makes things really difficult to understand ... Used properly and used reasonably sparingly, it can be really advantageous."

(This reiterates the conclusions drawn in [Ponder and Bush, 1994] and [Wilde and Huitt, 1992]). (Tables B.20 and B.21 in Appendix B provide individual comments).

Dynamic binding. Similarly, regarding dynamic binding, while every subject considered it as a clearly defined benefit (subject A remarked "without it there is no

object-oriented programming") subjects C, E, L, and M felt that there can be understanding difficulties if the programmer was "careless" when using it. Subjects C and L mentioned that one of the big problems is the calling of non-existent methods. If managed carefully, however, according to subject D, "it simplifies the code ... It's a more intuitive way of working." (Table B.19 in Appendix B provides individual comments).

Software reuse. All subjects, except subject E made frequent use of commercially produced libraries, but subjects A, D, I, K, and L were the only frequent users of in house (local) class libraries. No generalisation can be made, but it does support asking the question is the object-oriented approach meeting its reputation for ease of software reuse? (Tables B.13, B.14, and B.15 in Appendix B provide individual comments).

Overall benefit. Subjects were asked to complete the interview with a conclusion on the utility of the object-oriented paradigm (is it beneficial?). Nearly all the subjects were very positive about the object-oriented paradigm — the main benefit being the ability of the paradigm to map real world domains into software. Subjects regarded the technology as being preferable to work with, saying that it makes life easier and makes programming "good fun". One subject commented that it is beneficial where it is appropriate (some types of problem are not suited to an object-oriented solution). The only major drawback was seen to be the fierce learning curve which caused initial difficulties in understanding aspects of the technology (e.g., polymorphism and dynamic binding).

7.5 Conclusions

Interviewing users on the advantages and disadvantages of aspects of software engineering can be rewarding: it can help to identify (i) problems not yet considered, (ii) conditions under which existing problems are compounded, alleviated, or do not apply, and (iii) issues which appear worthy of further empirical inquiry and those that do not. The structured interview is an effective method of conducting a primary investigation within a multi-method programme of empirical research. Although the results of this study cannot be generalised with a great degree of confidence, the opinions gathered provide pointers to areas of interest and potential problems.

It is believed that the weaknesses of the structured interview technique were not a major influence on this study because (a) anonymity of subjects and the interview

setting encouraged them to speak freely, (b) the interviewer's experience in the domain meant further investigation of important points was conducted as required, and (c) clarification of elicited knowledge could be made after the interview transcript was available (although this was required from only one subject). The analysis was performed by transcribing and summarising each interview and tabulating subjects' answers to each question. Interview transcripts were read at least several times and it is unlikely that any important points were overlooked. Analysis of the collected interview data uncovered several aspects of marked interest:

Learning curve. The learning curve was mentioned as being steep when switching to the object-oriented paradigm from another, different paradigm — one subject mentioned the figure of almost two years to make the transition. Note well: this conclusion is based mainly on information elicited from subjects who were not daily users of object-oriented technology. It may also be contradicted somewhat by the fact that a subject with only six months experience was a very knowledgeable and proficient developer.

Inheritance. Depth of inheritance was discussed: excessive inheritance depth or inappropriate use of inheritance can cause understanding difficulties and is therefore something to be avoided (any initial time saved will be outweighed by the difficulties the inappropriate inheritance causes). In contrast, inheritance, when designed and implemented appropriately, is more likely to aid understanding than compound it. Understanding difficulties are created by dependencies caused by inheritance, however. Tracing a line of dependencies through inheritance hierarchies to find where the work is being performed can be a time-consuming task.

Design documentation. Object-oriented systems are as prone, if not more so, to the problems of missing documentation. Several subjects mentioned that time constraints can cause documentation to suffer. The drawbacks of missing documentation should be considered carefully from a maintenance perspective. If the documentation captures the correct information it could reduce future time pressures: subsequent changes may be properly designed, reducing system degradation; if not then the reverse might be true.

Design. The design of an object-oriented system appears to be extremely important: subjects continually mentioned design when discussing the problems of inheritance, software maintenance, polymorphism, and dynamic binding, stating that

if the design was appropriate then understanding was not constrained. In contrast, however, if not well designed then the system could be a "nightmare" to understand.

Software maintenance. As a consequence of software maintenance, object-oriented systems are still prone to degradation. The consensus was that object-oriented software would resist software degradation through maintenance better than other software systems, but only if the object mindset is kept when performing maintenance: 'quick fixes' are likely to increase software degradation (or the rate of entropy, as referred to by 2 subjects) to that of any other software. Trainees, therefore, are not the best people to perform maintenance.

C++. Subjects indicated a lack of enthusiasm towards C++ and viewed it inferior in many aspects to 'purer' object-oriented languages.

Conducting the structured interviews was the first phase of the multi-method programme of research. The next phase of the programme of research involved the design of a questionnaire from the structured interview findings to gather quantitative data from a larger sample of the object-oriented practitioner population.

Chapter 8

Phase II: A Questionnaire Survey

8.1 Introduction

This chapter presents the findings of a questionnaire survey, the second phase of the multi-method programme of research. The questionnaire survey followed the structured interviews presented in Chapter 7 with the intention of examining their findings on: (i) the perceived advantages of the paradigm, (ii) inheritance, (iii) the difficulties object-oriented code can cause, (iv) software maintenance and its consequences, (v) use of in house (local) class libraries, and (vi) the C++ programming language. A survey of this nature is appropriate as the second phase within the programme of research because questionnaires have the advantage of being able to collect large amounts of data which do not suffer from interviewer bias. The utility of questionnaires as a technique for empirical software engineering research is discussed, in particular the use of electronic newsgroups as a medium for questionnaire distribution (Section 8.2). The design and distribution of the questionnaire which was posted to appropriate world-wide electronic newsgroups and to members of a U.K. postal object-oriented mailing list is fully detailed in Section 8.4. The remainder of the chapter summarises the collected data and presents in depth analysis.

8.2 Using questionnaires as an empirical technique

There exist various media for distributing questionnaires, e.g., electronic newsgroups, face-to-face, magazines, post, the telephone (see Table 1 in [Miller *et al.*, 1996] for an overview of the comparative advantages and disadvantages). In common with all empirical techniques, questionnaires have both strengths and weaknesses:

Strengths: In comparison to other empirical techniques requiring an arbitrary number of subjects, questionnaire surveys are relatively inexpensive to conduct.

Data from questionnaires can be collected reasonably quickly because it is the respondent who completes the process on their own unlike other techniques which require skilled intermediates to record the process.

Questionnaires do not suffer the problem of interviewer bias: the responses given are not affected by the pressures an interview can cause (as discussed in Chapter 7).

If good response rates are achieved from a representative sample then findings can be generalisable.

Questionnaires can be completed anonymously and confidentially, providing obvious advantages, e.g., truthful responses.

Analysis of questionnaire data, while not easy, is not as difficult as analysis of structured interview or verbal protocol data. That said there are still difficulties, e.g., editing the collected data into the format required by a software analysis package is both time consuming and error prone (Kikuchi *et al.* discuss using neural networks as a possible method of counteracting this error proneness [Kikuchi *et al.*, 1993]).

Weaknesses: According to Sinclair, designing and administering a high quality questionnaire is a skilled task; a specialist in the behavioural sciences, a computer expert, and a statistician working as a team may be required to complete the process [Sinclair, 1990].

It may be difficult to sample the population in a representative way creating bias in the results. Poor response rates may also bias the results.

If the information obtained is not factual or easily checked then its reliability and validity are not always guaranteed [Goguen and Linde, 1993]. For example, smokers answering a medical questionnaire may be inclined to be less than honest about the number of cigarettes they smoke for fear of being morally criticised.

There is little scope to correct misunderstandings or probe responses once a questionnaire has been returned.

Using questionnaires is an excellent method of providing large amounts of quantitative data from a population under investigation relatively cheaply and efficiently. However, they are not a panacea and it can be difficult to produce generalisable findings. Consequently, it is argued that important findings are tested further by other empirical means.

8.2.1 Postal methods as a medium

Using postal methods to conduct a questionnaire survey is a popular medium primarily due to financial cost: all that is required is the photocopying costs, sampling costs, address labels, and mailing costs. In comparison to other questionnaire techniques, postal methods offer the following

Advantages: Using postal methods as a medium provides good access to the population sample, and also means the sample are likely to read the questionnaire.

The use of reminders and self-addressed stamped envelopes for return can help increase the response rate (although they can also add significantly to the survey costs), reducing the bias of non-response.

Speed of turnaround can be reasonably efficient if the urgency of the survey is stressed in the covering letter.

The respondent may consult others before completing the questionnaire, or pass the questionnaire on to a more knowledgeable person (although this can be viewed as a disadvantage for certain surveys).

Disadvantages: The questionnaire usually has to be kept reasonably short: lengthy questionnaires may discourage response.

The ability to ask open questions is very restricted because such questions require writing, and respondents generally tend to leave these questions blank or give a two or three word answer. Also, open questions usually require probing the answer given and this cannot be performed.

Finally, postal survey methods generally suffer poor response rates which may bias the findings.

Much has been written about postal survey methods by other researchers, e.g., see [Dillman, 1978], [Oppenheim, 1992] and [Sudman and Bradman, 1983] for more detail.

8.2.2 Electronic newsgroups as a medium

No relevant literature has been found on the utility of electronic newsgroups as a medium for distribution of questionnaires. In the author's experience of conducting an electronic survey similarities appear to exist with the utility of Magazine Distributed Questionnaires (MDQs). Using electronic newsgroups to gather information is an appropriate empirical technique subject to criteria similar to those stated by Pratto and Rodman for MDQs: (a) the target population is defined in terms of some phenomenon that is not widespread in large populations and there exists electronic newsgroups that appeal to the target population, (b) there is no commonly available sampling frame[1] for the target population, and (c) investigative research on the topic is limited [Pratto and Rodman, 1987]. Once satisfied, using Electronic Newsgroup Distributed Questionnaires (ENDQs) can provide the following:

Advantages: ENDQs provides very quick access to members of the target population, i.e., by making use of the vast number of existing electronic newsgroups it is possible to select the ones which have a large proportion of individuals who are part of the target population. In fact targeting is highly efficient using this method, as newsgroups are in general very focused and are unlikely to be read by anyone not interested in the topic. Also these newsgroups can provide access to specialised populations, which are often unavailable in other media. This again is due to the diversity of the topics under discussion across these groups.

The monetary cost of such a survey is exceptionally low, an advantage when performing exploratory research with limited financial support. This type of survey usually works on a something-for-something basis: the individual is asked to respond, and, in turn, is promised access to the survey results (usually posted to the corresponding newsgroups).

ENDQs have the ability to obtain a substantial amount of quantitative data: popular electronic newsgroups are read by thousands of individuals from all over the world.

The speed of turn-around from beginning the ENDQ to collecting the data is quick because sampling frames are not needed, i.e., once the questionnaire has been completed, all that is required is posting it to the appropriate newsgroups (responses can be returned within a matter of hours). This is also an advantage

[1] A sampling frame is a list of all of the members of the population under investigation from which the sample is drawn.

for planning future research with deadlines, e.g., subject-based experiments require advanced laboratory bookings, subject recruitment, and so on, and ENDQs can quickly help to focus hypotheses for testing.

Disadvantages: A questionnaire posted to electronic newsgroups will be distributed to a 'biased' population: only those who subscribe to the newsgroup can read the questionnaire.

Respondents to the questionnaire are self-selected from the total population for their motivation and interest to respond. Also, the most effective method of increasing response rates in postal questionnaires is through follow-up contacts: this method is not available for ENDQs for obvious reasons and demonstrates further the problem of self-selection. Self-selection is a problem which, to a certain degree, exists for all methods of questionnaire distribution [Hawkins, 1975], although for ENDQs the problem is pronounced.

Finally, the ability to ask open questions is still as restricted as in postal method questionnaires because such questions require typing. Open questions require probing of the answers given and this still cannot be easily performed.

A full discussion of the utility of ENDQs is provided in [Miller *et al.*, 1996].

8.2.3 Conclusions

To conclude, researchers should be aware of the advantages and disadvantages of using questionnaires before deciding to place any significance on their findings: when using ENDQs there is the likely bias of self-selection. Using ENDQs also offer various advantages, notably the collection of data relatively quickly and cheaply. This data can then be used to form specific hypothesis to be tested by more rigorous empirical techniques. Postal surveys, on the other hand, produce results that can be representative of the population under study assuming a sampling frame is available and a reasonable response rate is achieved. They are also more expensive and time consuming to perform in comparison to an ENDQ survey.

8.3 Designing a questionnaire survey

A questionnaire usually consists of the following sections: (i) a prologue which introduces the topic and attempts to motivate the reader to respond, (ii) a classification section asking for personal details such as name, age, experience, (iii) an information

section detailing the questions on the phenomenon under investigation, and (iv) an epilogue which thanks the respondent for participating and supplies instructions for returning the questionnaire [Oppenheim, 1992].

Although it is sensible to construct the questionnaire in this format, eliciting accurate information is primarily dependent on the design of the questions. First, before deriving the questions, Sinclair states the following points require answers: (i) what are the results supposed to show (what are the objectives of the study), (ii) what level of accuracy is required, and (iii) what additional data is required to link this survey to other research [Sinclair, 1990]. Without these answers the questionnaire begins with vague thinking, which produces vague questions, which ultimately, regardless of how good the analysis, produces vague answers. Sinclair concludes that there is no substitute for this part of the design. Second, the language used to phrase the questions should be carefully chosen. It should be understandable and unambiguous, i.e., the question should mean the same to everyone regardless of the context the respondent uses it in, although the validity of this position has been questioned [Goguen and Linde, 1993]. Third, questions should be as short as possible, and not double-barrelled, e.g., it can be hard to analyse a question such as 'Are you familiar with structured design or object-oriented design?' Fourth, questions should not be loaded in a particular direction, i.e., they should not invite any particular answer to be given. Finally, respondents may pass off their opinion as truth creating bias in the results. If this is a significant worry, it can be catered for by the use of closed questions[2] which offer the advantages of: (i) clarifying the alternatives for the respondent thus reducing the likelihood of snap responses, (ii) ease of analysis, and (iii) helping to eliminate the useless answer, e.g., 'How long have you been writing object-oriented code? Since I started programming in C++'. Closed questions, however, also have several disadvantages: (i) they must cover the total response range, (ii) they create a forced-choice situation, (iii) all the answers must seem equally attractive, and (iv) in difficult questions they allow the respondent to hide in the safety of the 'don't know' answer.

Once the questionnaire is complete, it is important to reduce the bias of the sample respondents, i.e., the respondents should be representative of the population under investigation; this is best achieved by randomly sampling the population as far as possible. This is a difficult task and requires an available sampling frame for the population: it is therefore impossible to have random sampling in ENDQs. A problem

[2]Closed questions provide the answers from which the respondent must choose. Open questions, on the other hand, require the respondent to provide their own answer.

with non-representative samples is that the findings cannot be easily generalised. For more detailed discussion on questionnaire design (and other important survey aspects) see [Biemer *et al.*, 1991] and [Oppenheim, 1992].

8.4 Questionnaire construction and administration

8.4.1 Layout

The layout of the questionnaire followed the guidelines detailed in Section 8.3. A letter[3] introducing the topic of the questionnaire, the motivation for conducting the survey, and the instructions for returning completions was considered the prologue section. The classification section collected details on the respondents' position at work, experience with the object-oriented paradigm, familiarity with object-oriented languages, and information on experience with inheritance, maintenance, and class libraries. The information section asked questions to which the answers were more based on opinions derived from "experience, reading, or conferring with colleagues." The epilogue allowed the respondent to make additional comments and elaborate on points made in the questionnaire, and thanked the respondent for participating (see Appendix C for a copy of the distributed version of the questionnaire).

The principle concern was to keep the layout differences between the two versions to a minimum, and this was achieved. Media dependent differences were noted after the questionnaire had been distributed, however (see Section 8.4.4 for details).

8.4.2 Derivation of the questions

As stated, the questionnaire survey is the second phase of the programme of research. The aim was to explore users attitudes to some of the findings of the structured interviews. Analysis of these attitudes can be used to form hypotheses to be tested using more specific forms of experimentation, e.g., a questionnaire devoted to object-oriented software maintenance, or a controlled laboratory experiment to test a smaller, but more focused hypothesis. Issues discovered from conducting the interviews lead to the following survey objectives:

1. Explore attitudes to the perceived advantages of ease of analysis and design, programmer productivity, software reuse, and ease of maintenance relative to other paradigms.

[3] At the beginning of the file in the electronic version; a separate sheet of paper in the postal one.

2. Explore attitudes towards inheritance (including depth of inheritance and multiple inheritance) and measure how often reported difficulties in the literature occur in reality.

3. Catalogue the difficulties of understanding object-oriented code.

4. Explore attitudes to object-oriented software maintenance and its consequences.

5. Discover if the promise of software reuse is being met only through commercial class libraries or if in house (local) class libraries are being used.

6. Explore practitioners attitudes towards the C++ programming language.

Nineteen questions (several with two or more parts) were derived from these objectives (this includes three initial questions on experience, job classification, and object-oriented language familiarity). No more than four questions were asked on any issue. To facilitate ease of analysis all questions, except one, were closed although extra written information was encouraged. The categories provided covered the range of possible answers through either a nominal data category or an ordinal data range (range of 1 to 5, where 1 = Never and 5 = Always). The only open question, based on objective 3 above, was considered too important to create a forced choice situation. Another concern was that a closed alternative may not cover the entire response range.

8.4.3 A pilot study

Piloting of a questionnaire is important to discover unnoticed assumptions in questions such as loading biases, ambiguities, or unnecessary complexity. If these are not identified before the questionnaire is distributed little can be done to rectify the situation [Oppenheim, 1992].

A draft questionnaire was posted to a local newsgroup where it was answered by fellow researchers as meticulously and pedantically as possible. The comments were collated and led to the conclusions:

1. It was obvious that some of the questions required either clarification or simplification, e.g., "too vague", "will need to explain this", "what is this trying to find out?"

2. It was not clear whether respondents could mark more than one box off in the closed questions, e.g., "can I put more than one X here too? Your instructions are not clear."

3. Some of the questions were phrased in a loaded or biased way, e.g., "Is this too directed/loaded?", "Biased?"

4. Closed questions did not cover the full response range: several respondents gave answers outwith the range provided.

The identified faults were rectified and the revised version was reviewed a second time. Several comments lead to minor changes, after which the questionnaire was distributed. Note that the initial responses were not included in the data analysis.

8.4.4 Distribution

Four hundred questionnaires were randomly posted to members of a mailing list (consisting of 2000 names and addresses) within the U.K. All members of the list had in the past expressed interest in the object-oriented technology of a certain software company. Unfortunately, nothing can be said about how representative recipients of the questionnaire were because it was not known how representative of object-oriented practitioners the mailing list was. The responses received, however, were from a variety of practitioners, from a wide number of academic and industrial organisations across the country.

The questionnaire was also posted to the world wide distributed electronic newsgroups comp.databases, comp.lang.clos, comp.lang.c++, comp.lang.eiffel, comp.lang.objective-c, comp.lang.smalltalk, comp.object, comp.software-eng, and comp.sys.next.programmer. This approach enabled a variety of different people access to the questionnaire, e.g., software engineers, project managers, academics. As discussed in Section 8.2, this approach suffers from self-selection: only individuals who subscribed to the newsgroups were able to read it and only those motivated enough may have responded. Subsequently, the respondents may not be representative of the object-oriented population. The responses received, however, did come from various groups of people from all over the world.

It is important to note that it is extremely difficult, if not impossible, to obtain a representative sample of the object-oriented practitioner population because there is no sampling frame available. Consequently, although little can be said about the sample being representative of the population, there can be more confidence in the findings if no difference of opinion exists between the two sets of respondents.

Distribution differences

The experience of using different media to conduct the survey uncovered a number of differences between them. An aspect not considered during the questionnaire design was the ability to edit the questionnaire: many respondents of the electronic version increased the space allocated for comments after each question, while postal respondents rarely appended further information on extra sheets of paper. It would seem the electronic version facilitated extra written comments, although its respondents may just have been more motivated (as discussed in Section 8.2). Another significant difference was the access to members of the target population: it was exceptionally easy to repost the questionnaire to the electronic newsgroups to attract more responses, but, because of financial limitations, reminders could not be sent out to members of the mailing list. These differences should be classified as significant benefits for using electronic newsgroups over postal methods (assuming the postal sampling is not representative of the population, as is likely in this survey). Using both media, however, has the advantage of increasing the number and variety of survey respondents.

8.5 Responses received

The electronic survey received 167 responses to the questionnaire. The postal survey received 155 returns, although 11 of these were too incomplete to be included in the analysis and 36 were marked return to sender. Using the formula

$$(\text{number returned} \,/\, (\text{number in sample} - (\text{non-eligible} + \text{non-reachable}))) \times 100$$

provides a response rate of at least 33%. McNeill notes that response rates for mail-based questionnaires are usually in the region of 30-40% [McNeill, 1985]. According to Edwards, a response rate of 20-30% for mail-based questionnaires can be considered adequate [Edwards, 1972]. In their well referenced software maintenance survey which included sending reminders, Lientz and Swanson 'only' produced a 24.6% response rate [Lientz and Swanson, 1980]. Dillman remarks that use of the total design method (TDM) can help achieve response rates between 60-75% [Dillman, 1978]. Among other attributes, TDM requires a sampling frame of the population under study and a mechanism for contacting questionnaire recipients who have not responded. Such a high response rate was not achieved for this survey for two reasons. First, a proper sampling frame was not available — the mailing list was not a list of people who used object-oriented technology. Consequently, many recipients who were unable to complete the

	Survey					
Position	Electronic	Electronic %	Postal	Postal %	Total	% of Total
Student	34	20.4	3	2.8	37	13.5
Academic	23	13.8	11	10.2	34	12.4
Software engineer	85	50.9	42	38.9	127	46.2
Project manager	10	6.0	25	23.1	35	12.7
Other	15	9.0	27	25.0	42	15.3

Table 8.1: Respondents' positions

questionnaire may have simply discarded it. An estimate of this number cannot be provided, but it is likely to be reasonably high (which would increase the response rate considerably). Second, this survey was conducted anonymously — reminders could not be sent to recipients who had not returned the questionnaire.

Before discussing the analysis of the 275 completed questionnaires information is presented on respondents' attributes. Table 8.1 provides a breakdown of the different positions held by respondents, the largest proportion being software engineers (46.2%).[4] The 'Others' category consists of class librarians, consultants, research assistants, system managers, and technical directors. Table 8.2 presents a break down of the respondents' experience of the object-oriented paradigm, the largest proportion (34.4%) having greater than 4 years experience. Further, approximately 90% of the respondents reported to be using object-oriented technology more than twice every working week, with only 2% using it less than once a week. Finally, a break down of the languages respondents are familiar with is presented in Table 8.3. The table displays the number of respondents familiar with C++, Objective-C, and so on, and the percentage this accounts for of the total respondents. The 'Others' row contained the languages C-Flavors, Dylan, OO COBOL, OO Pascal, Sather, and Simula. As expected, C++ is the most familiar language overall although the percentage of respondents familiar with it might have been higher given the publicity it receives. Further examination of the dataset found that software engineers (67%) were more

	Survey					
Time (years)	Electronic	Electronic %	Postal	Postal %	Total	% of Total
< 1	8	4.8	17	16.0	25	9.2
1 - 2	39	23.4	30	28.3	69	25.3
3 - 4	58	34.7	27	25.5	85	31.1
> 4	62	37.1	32	30.2	94	34.4

Table 8.2: Respondents' experience with the OO paradigm

[4] All percentages have been calculated using the number of responses received for each question. The total number of responses to each question can be found from the appropriate table in Appendix C.

familiar with C++ than were academics (50%) or project managers (37%).

	Survey					
Language	Electronic	Electronic %	Postal	Postal %	Total	% of Total
C++	122	73.1	37	34.3	159	57.8
Objective-C	57	34.1	41	38.0	98	36.6
Eiffel	14	8.4	3	2.8	17	6.2
Smalltalk	57	34.1	36	33.3	93	33.8
CLOS	15	9.0	0	0.0	15	5.5
Other	14	8.4	35	32.4	49	17.8

Table 8.3: Respondents' familiarity with different OO languages

8.6 Analysis and discussion

Analysis was performed by tabulating the responses for each questionnaire into SPSS format which was then used to calculate frequencies and percentages (presented in tabular form in Appendix C). Statistical tests were applied where appropriate.

8.6.1 The OO paradigm in comparison to other paradigms

(Drawn from Q. 10(a), (b), (c), and (d) in Appendix C. Summary statistics are presented in Tables C.10, C.11, C.12, and C.13).

Many unsupported claims about the benefits of the object-oriented paradigm over other paradigms have been made, particularly with respect to ease of analysis and design, programmer productivity, software reuse, and ease of maintenance. Figure 8.1 displays the distribution of responses to a closed question on these issues. The results show a huge positive response for the object-oriented paradigm in each of these categories. A subset of respondents qualified their response: *The problem:* not all problems are best suited to an object-oriented solution, *Design:* if the design is poor then an object-oriented software system is likely to be more difficult to maintain than a poorly designed conventional equivalent, *Quality of abstraction:* if hidden assumptions are made during the implementation, resulting software will be harder to reuse, and *All things being equal between the paradigms:* e.g., a skilled non object-oriented programmer will be more productive in their paradigm than a non skilled object-oriented programmer. While these views are plausible, the fact remains the large majority of respondents view the object-oriented paradigm as offering benefits in these four categories (remembering the sample are very probably object-oriented advocates).

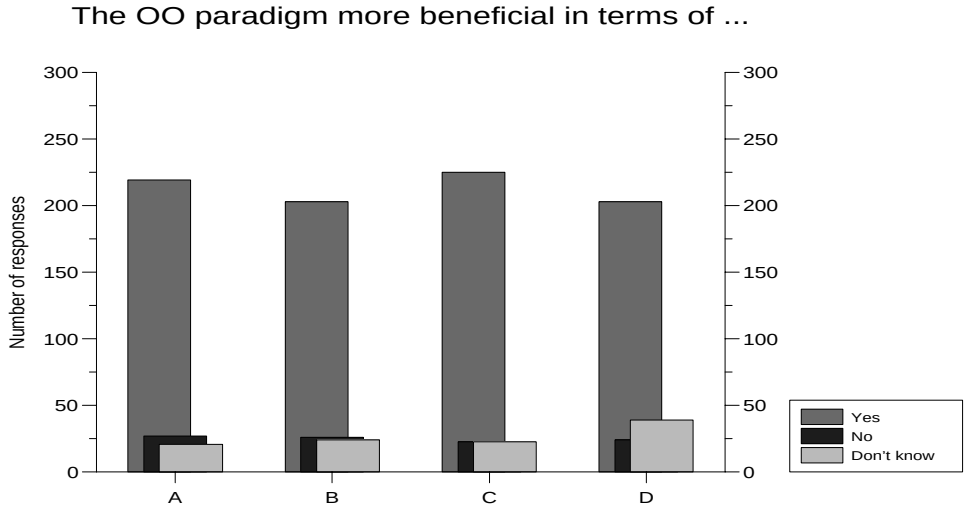


Figure 8.1: Is the object-oriented paradigm more beneficial than other paradigms in terms of A: ease of analysis and design, B: programmer productivity, C: software reuse, and D: ease of maintenance?

8.6.2 Inheritance

(Drawn from Q. 5, 7, and 11 in Appendix C. Summary statistics are presented in Tables C.6, C.7, and C.14).

Three questions were devoted to the topic of inheritance: First, depth of inheritance was considered. A closed question asked at which depth does inheritance begin to cause understanding difficulties. The data, illustrated in Figure 8.2, shows that of those respondents that felt depth of inheritance causes difficulties (approximately 55%), the largest proportion marked 4-6 levels of inheritance as the region where the problems start. A Chi-square test performed on the respondents who reported having problems with depth of inheritance against those who did not, provided a statistical significant result, $\rho < 0.05$ (one-tailed, $df = 3, X^2 = 6.79$), for an association between experience and having a problem with depth of inheritance, i.e., the more experienced the developer the less likely they are to have a problem with depth of inheritance. Yet the frequency of experienced respondents who reported a problem with depth is still high: 43 respondents (50.6%) with 3 - 4 years experience and 43 respondents (45.7%) with $>$ 4 years experience. In addition, a Chi-square test was calculated to check if having a problem with depth of inheritance was language dependent, but found no statistical difference (two-tailed, $df = 2, X^2 = 2.54$). Regardless, the response distribution shown in Figure 8.2 indicates that inheritance depth can cause understanding problems. Chidamber and Kemerer [Chidamber and Kemerer, 1994] discuss depth of inheritance as a metric and present data on this from libraries of two different sites: at site A (C++ library) only approximately 75 from 634 classes have a depth of 4 or more (median=1,

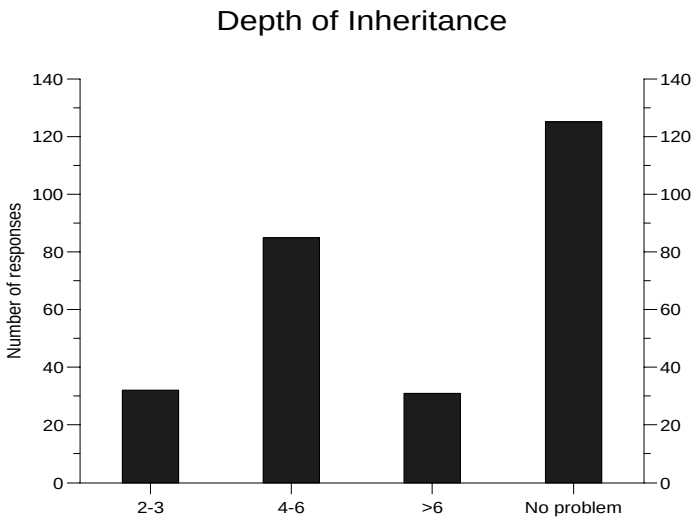


Figure 8.2: Depth at which inheritance begins to cause difficulties

max=8). At site B (Smalltalk library) approximately 550 from 1459 classes have a depth of 4 or more (median=3, max=10). Similarly, Miller *et al.* present a mean depth of 1 and a maximum of 4 for their inspected C++ code [Miller *et al.*, 1994]. Viewing these figures with the presented data asks the question: are programmers deliberately avoiding the creation of deeper hierarchies or are shallower hierarchies just a more natural model?

Second, respondents were asked to grade how often inheritance caused them understanding difficulty. It was generally agreed that inheritance does cause understanding difficulties: only 19% of respondents said it *never* caused any difficulty. This is not significant because the largest proportion (48.5%) said it caused difficulty only occasionally. What has significance is that when asked what has caused the most difficulty when trying to understand object-oriented software, inheritance was the second most popular answer after missing or inadequate design documentation (see Section 8.6.3). As discussed in Chapter 6, Section 6.3 related research has reported that inheritance can cause:

- Distributed class descriptions through the hierarchy.
- Class-to-class dependencies.
- Understanding and maintenance difficulties.
- Increased code complexity.

These inheritance related difficulties were specifically mentioned by many respondents.

Third, respondents were asked to grade the usefulness of multiple inheritance (although a subset of the sample may not have been familiar with this concept because

not all object-oriented languages support it, e.g., Objective-C). The distribution of responses was spread relatively evenly across the middle categories, with minorities of 11.7% and 10.5% reporting it was never and always of use respectively. The utility (and concept) of multiple inheritance may be language dependent, but a Chi-square test did not indicate statistical significance (two-tailed, $df = 8, X^2 = 9.00$). According to Perry and Kaiser multiple inheritance is widely recognised as both a blessing and a curse and the response distribution supports this position [Perry and Kaiser, 1990]. The argument of multiple inheritance versus single inheritance will continue: some arguing that it produces a more complex design, is more difficult to test, is more difficult to reuse, and is easy to abuse; others arguing it maps the reality of the domain being modelled producing a more appropriate design, and it facilitates software reuse and maintenance. While multiple inheritance may be more complex than single inheritance the questionnaire data supports the theory that multiple inheritance is a useful concept. A source of concern, however, is that multiple inheritance is used when it is inappropriate. As a consequence, object-oriented software can become more complex than is necessary. One method of preventing unnecessary complexity may be through use of patterns (see, e.g., [Gamma *et al.*, 1995]).

Typical method size

(Drawn from Q. 4(a) and (b) in Appendix C. Summary statistics are presented in Tables C.4 and C.5).

Wilde *et al.* report that maintainers of object-oriented code often must trace through chains of dependencies created by inheritance, a problem compounded by the proliferation of small methods [Wilde *et al.*, 1993]. Small methods, however, have been advocated by much of the work on good object-oriented programming style, e.g., [Henricson and Nyquist, 1992], [Johnson and Foote, 1988]. The data summary in Table 8.4 displays the frequencies to respondents' typical method size (in executable lines of code) and the upper and lower limits of methods that programmers have written. The response distribution is something of a bell curve: a typical method appears to fall within 12 ± 7 lines of code. While a typical method is less than 20 lines of code long for nearly 90% of the respondents, 44% of respondents reported their largest methods exceeded 50 lines of code.

Table 8.5 presents the frequencies of typical method size for the most popular languages (any respondent who circled familiarity with more than one language could not be included in any of these columns owing to inability to distinguish which language

Typical method size	Frequency	%	Method range	Frequency	%
1 - 4 lines	20	8.0	1 - 50 lines	116	55.8
5 - 10 lines	101	40.2	1 - 100 lines	49	23.6
11 - 20 lines	104	41.4	1 - 150 lines	8	3.8
> 20 lines	26	10.4	1 - 200 lines	22	10.6
			1 - 250+ lines	13	6.3

Table 8.4: Frequency of typical method sizes and method ranges

was being described). Wilde *et al.* present an analysis of three software systems, reporting that 50% or more methods are fewer than 4 Smalltalk lines or 2 C++ lines independent of the application domain [Wilde *et al.*, 1993]. The data presented above does not seem to support this finding, but is more supportive of the data presented in Miller *et al.* who report a mean method size of C++ code for experienced programmers as 9 lines of code (range 1 - 35) [Miller *et al.*, 1994].

	Object-Oriented Language					
	C++	%	Objective-C	%	Smalltalk	%
1 - 4 lines	4	6.8	1	7.1	5	12.2
5 - 10 lines	19	32.2	5	35.7	20	48.8
11 - 20 lines	25	42.4	7	50.0	15	36.6
> 20 lines	11	18.6	1	7.1	1	2.4
Total	59		14		41	

Table 8.5: Language breakdown of typical method sizes

8.6.3 Difficulties in understanding an OO program

(Drawn from Q. 6 in Appendix C).

An open-ended question asked what causes the most difficulty when trying to understand an object-oriented program. The most frequently appearing answers were:

1. missing or inadequate design documentation (39, 16.8%),
2. inheritance (36, 15.5%),
3. poor or inappropriate design (including inappropriate use of OO concepts) (30, 12.9%),
4. tracing a line of method invocations to find the method which performs the work (20, 8.6%),
5. the C++ programming language (including syntax, languages obscurities, and the `friend` function) (17, 7.3%),

6. method naming confusions (including obscure and inconsistent naming) (11, 4.7%),
7. experience with the procedural paradigm before learning the object-oriented paradigm (9, 3.9%),
8. relationships between classes and how objects communicate (9, 3.9%),
9. polymorphism (8, 3.4%), and
10. understanding 'clever' coding styles (8, 3.4%).

Other less popular responses were dynamic binding, hybrid code (mixture of both object-oriented and conventional code), knowing what code to reuse from a class library, and the splitting up of a system into many small files. There were, however, 5 subjects that stated that nothing specific had caused them understanding difficulties.

The survey data supports the premise that missing design documentation is a major source of heartache to developers attempting to understand what a software system is doing, and how it is doing it. As Davis states,

> Design without documentation is *not* design. I have often heard software engineers say "I have finished the design. All that's left is its documentation." Can you imagine a building architect saying "I have completed the design of your new home. All that's left is to draw a picture of it"? [Davis, 1994].

Moreover, the data supports the premise that object-oriented systems are equally susceptible to missing design documentation and the difficulties it causes.

Inheritance is reported as the second most popular reason for causing understanding difficulties in object-oriented systems. Possible reasons were discussed in Section 8.6.2.

Finally, the data suggests that poor or inappropriate design is a major source of understanding difficulty. (In the context of object-oriented systems, poor design includes inappropriate use of object-oriented concepts, inappropriate abstractions, and unnecessary complexity). Further, design must be considered with reference to inheritance. A poorly designed hierarchy will compound the problems discussed in Section 8.6.2. This is not a surprising result, but it does strengthen the argument that the object-oriented paradigm is not a panacea: object-oriented systems must still be appropriately designed. Failure to do so severely affects their understandability.

Missing or inadequate design documentation and poor or inappropriate design (two of the first three in the above list) are not paradigm specific and are concerned with the deficiencies of current software engineering practice. Perhaps an improvement of current practice would offer more advantages rather than making the transition to the object-oriented paradigm.

8.6.4 Maintenance of conventional and object-oriented programs

(Drawn from Q. 12, 13, and 14 in Appendix C. Summary statistics are presented in Tables C.15, C.16, and C.17).

Three closed questions were asked about software maintenance. First, respondents were asked if continual maintenance of conventionally (i.e., structured) designed software would lead to unmaintainability. The largest proportion (43.7%) of respondents thought this would usually happen. Respondents were then asked the same question in the context of object-oriented designed software. The largest proportion (48.5%) of respondents thought this would occasionally happen. A Wilcoxon signed ranks (related) test was calculated to test for ordinal level differences between the two responses. The results show significance at $\rho < 0.01$ (12 respondents circled a number greater for the second question, 148 respondents circled a number less for the second question, and 70 respondents circled the same number for both questions; two-tailed, $N = 230, Z = -9.458$). Thus it is concluded that respondents regard object-oriented software less likely to lead to unmaintainability.

Further, to examine the attitude of practitioners regarding object-oriented software facilitating maintenance, respondents were asked directly whether object-oriented software was generally more maintainable than the equivalent conventionally designed software. The largest proportion of respondents (58.4%) circled category 4, i.e., they thought this would usually be true. This statistic, however, does not paint a complete picture: disclaimers explained that this would be true if the software was designed well; if not then object-oriented software will be more difficult to maintain because of more complex and less intuitive inter-relationships.

8.6.5 Software reuse through in-house (local) class libraries

(Drawn from Q. 9 in Appendix C. Summary statistics are presented in Table C.9).

It is argued that object-oriented software facilitates reuse. As a consequence, frequent use of in-house class libraries might be expected. The respondents were asked to grade their use of such libraries. Responses received were divided between infrequent

users of in-house class libraries (those that said they never or only occasionally used them 43.3%) and frequent users (those that stated they usually or always used them 42.2%). A Chi-square test found no significance difference (two-tailed, $df = 2, X^2 = 3.1$) between the language known and frequency of use of in-house class libraries. In this survey, therefore, it is concluded that use of in-house class libraries is language independent. Interpretation of the almost equal split is difficult without comparative figures for in-house software libraries for the structured paradigm. Examining the complete dataset, however, shows 75.4% of respondents make use of in-house class libraries at least occasionally: although almost one quarter of respondents never make use of in-house class libraries, it does appear that local software reuse is becoming more widespread.

A small subset of respondents, however, warned that time can be wasted trying to find existing code to reuse, or understanding what code there is in an attempt to reuse it — Wilde and Huitt have stated that for the reuse benefits of object-oriented software to be achieved it must be possible to locate the required code efficiently [Wilde and Huitt, 1992]. An area of growing interest is Christopher Alexander's notion of patterns and pattern languages, developed for describing architectural constructs and now borrowed by OOA and OOD, which it is argued may reduce these problems. Alexander states,

> each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice [Gamma *et al.*, 1995].

Patterns are discovered by experience, and are described in a manner that emphasizes potential reuse: the pattern can be used as a higher level building block for OOA and OOD. More information on Alexander's pattern concept is provided in the excellent book by [Gamma *et al.*, 1995].

8.6.6 C++ as the de facto standard object-oriented language

(Drawn from Q. 15, 16, and 17 in Appendix C. Summary statistics are presented in Tables C.18, C.19, and C.20).

Three questions were asked about the C++ language. Respondents were told only to answer these questions if they had relevant experience. First, a closed question

measured attitudes towards C++ becoming the industry de facto standard object-oriented programming language. The largest proportion, 143 respondents (60.9%), regarded it as 'bad', 42 respondents (17.9%) regarded it as 'good' (the remainder were indifferent (14.9%) or unsure (6.4%)). A subset of respondents explained the reasons for their answers:

Bad: all the problems of C exist in C++; very obscure syntax; many of its features have been 'hacked' together, e.g., multiple inheritance; it allows the writing of straight C, i.e., not forced to write OO code; easy to override data security and break encapsulation, e.g., friends; almost encourages the programmer to take short cuts; has blackened the reputation of the object-oriented paradigm.

Good: efficient in comparison to purer OO languages; can make use of existing C libraries; many developers know C so the transition to C++ is easier and less costly; C++ has given many exposure to the object-oriented paradigm and as a consequence object-oriented programming has become mainstream.

A second question measured attitudes towards C++ allowing a mixture of object-oriented and conventional programming. The majority, 108 respondents (45.8%) regarded this a disadvantage, 86 respondents (36.4%) regarded it an advantage, and 25 respondents (10.6%) decided it was both. Again, a subset of respondents explained their reasons:

Disadvantage: leads to a mess of mixed metaphors which is harder to read and understand; for full encapsulation and ease of maintenance, software should be consistently designed in one paradigm, not two; allows for the sloppy approach because it enables reverting to conventional programming if the programmer runs into difficulties (especially novice OO programmers); complexity is increased; enables a gradual migration to OO which impedes the thought process; can claim that OO code is being written when it is not.

Advantage: OO is more appropriate at the design level — at the implementation level it helps to be able to perform conventional decomposition of methods; it allows freedom to choose the most appropriate technique; not all problems facilitate an OO solution, but the environment can be used regardless; transition for programmers with a conventional background can be made gradually, making it easier and maintaining programmer productivity.

Third, in the context of performing maintenance, respondents were asked about making use of the `friend` function[5] to prevent redesigning the inheritance hierarchy. 69 respondents (38.1%) said they would make use of a `friend` function in this manner only occasionally; further 45 respondents (24.9%) said they would never make use of a `friend` function to do this. Of concern, however, was the frequency of respondents who said they would use a `friend` function to prevent this more than just occasionally: 37 respondents (20.4%) said sometimes, 24 respondents (13.3%) said usually, and remarkably, 6 respondents (3.3%) said always. Stroustrup states that `friends` should only be used to avoid (a) global data, (b) global (non member) functions, and (c) public data members [Stroustrup, 1991]. Unfortunately, the `friend` function is easily abused; apparently programmers are using it to cut corners by breaking encapsulation enabling another class to directly manipulate the private data. Subsequent testing and maintenance becomes more difficult because class relationships become more obscure and more complex.

8.6.7 Questionnaire media differences

As discussed earlier in Section 8.2, a concern about the use of electronic newsgroups as a medium for questionnaire distribution is the problem of self selection — only those who subscribe to the newsgroup can read the questionnaire and respondents are self-selected from this 'biased' population for their motivation and interest to respond. It was therefore necessary to examine the data for evidence of self-selection. If strong relationships are detected between a survey medium and respondents' answers then this would constitute such evidence.

A Chi-square test was performed on each question variable to check for answer differences between the electronic newsgroup respondents and the mailing list respondents. Table 8.6 presents the results of each test where any result indicated as significant is at the $\rho < 0.05$ level or better. The significant result achieved for respondent position, Q. 1(a), demonstrates that in this survey different media have targeted different cross-sections of the population (something discussed as a distinct possibility in Section 8.4.4). The significant differences achieved for capacity used, experience, and language familiarity are not independent of this fact and can therefore be easily explained: because two different groups of practitioners were questioned it is understandable that differences exist (i) in the capacity respondents use the paradigm, (ii) in respondents' object-oriented experience, and (iii) in respondents' familiarity with

[5] `Friends` of a class are trusted with access to the private and protected members of that class.

Description	Question	Degrees of freedom	Test result (X^2)	Significant?	Cramér's ϕ
Position	Q. 1(a)	4	44.08	Yes	0.40
Capacity used	Q. 1(b)	3	8.46	Yes†	0.13
Object-oriented experience	Q. 2(a)	3	12.28	Yes†	0.21
Language familiarity	Q. 3	5	50.86	Yes	0.34
Typical method size	Q. 4(a)	3	2.59	No	0.10
Method size range	Q. 4(b)	4	7.81	No	0.19
Depth of class hierarchy	Q. 5	3	4.37	No	0.13
Inheritance caused difficulty	Q. 7	4	2.00	No	0.09
Problems caused by naming	Q. 8	4	4.87	No	0.14
Use of local class libraries	Q. 9	4	8.63	No	0.18
Ease of analysis and design	Q. 10(a)	2	10.53	Yes†	0.20
Programmer productivity	Q. 10(b)	2	5.68	No	0.15
Software reuse	Q. 10(c)	2	0.77	No	0.05
Ease of maintenance	Q. 10(d)	2	1.49	No	0.07
Multiple inheritance useful	Q. 11	4	5.88	No	0.15
SP maintenance problems	Q. 12	4	2.42	No	0.10
OO maintenance problems	Q. 13	4	4.36	No	0.14
OO more maintainable than SP	Q. 14	4	6.25	No	0.16
C++ as a de facto standard	Q. 15	2	2.62	No	0.10
C++ allows hybrid programming	Q. 16	3	3.01	No	0.11
C++ `friend` function	Q. 17	4	3.31	No	0.14
Use operator overloading	Q. 18(a)	4	2.01	No	0.10
Overload operators as	Q. 18(b)	3	4.43	No	0.17
Use of templates	Q. 19	4	2.94	No	0.13

Table 8.6: Two-tailed Chi-square test results between each variable and the media used for the questionnaire distribution. †— see text for explanation

different object-oriented languages (although capacity used and experience are not significant under the Bonferroni correction method discussed below).

Although statistically significant relationships have been discovered by the Chi-square test, this significance does not indicate the strength of the relationship: a significant result only means that the relationship in the population is unlikely to be zero [Welkowitz *et al.*, 1976]. It is, therefore, desirable to have a measure of the strength of the relationship, i.e., have an index of the degree of correlation. For this reason Cramérs ϕ was calculated, a linear index which converts the Chi-square X^2 value to a correlation coefficient (interpreted as a Pearson r correlation coefficient) indicating the strength of the relationship between two different variables. The index is on the scale of 0 to 1 where the larger the value of ϕ, the stronger the relationship between the two variables. Note that the largest value, achieved for position ($\phi = 0.40$), only represents a weak to moderate relationship between the medium used and the position the respondent held. Consequently, given that (i) of the remainder of the questions tested (Q. 4 to 19), with the exception of Q. 10(a), provided no significant difference of opinion between the two sets of respondents and (ii) the calculated indexes for

these questions are indicators of no more than weak relationships, there is increased confidence that the self-selection problem has not biased the results of this survey.

The exception was a statistical difference of opinion about the ease of OOA and OOD (see Figure 8.1). The complete data set was examined in an attempt to explain the difference. A second Chi-square test was performed including only those respondents who chose either the 'yes' or 'no' category (to eliminate the 'don't know' respondents as the reason for the difference). A significant result was still obtained, $\rho < 0.05$ (two-tailed, $df = 1, X^2 = 5.94$). Further examination of the data set revealed that the significance was caused by the number of respondents who replied 'no' to the question: only 11 (7.2%) respondents in the electronic group compared to 16 (17.2%) in the mail group. The questionnaires for these respondents were then examined for similarities which might explain why they answered 'no', e.g., were they relatively inexperienced object-oriented practitioners? Cross checking across all the other variables in Table 8.6, however, did not reveal any common ground and meant no explanation could be based on the data. One possible reason of statistical significance may be because when conducting multiple comparisons, the probability of committing a Type I error increases with the number of tests. Courtney and Gustafson state, "although the probability of a Type I error is fixed at $\alpha = 0.05$ for each individual test, the probability of falsely rejecting at least one of those tests is significantly larger than 0.05" [Courtney and Gustafson, 1993]. Given that 24 statistical tests were applied, therefore, the probability that the significant result for Q. 10(a) was achieved by chance is quite high. If the answer to any question is independent of all other answers then this probability is calculated as $P(x \geq 1) = 1 - (1 - 0.05)^{24} = 0.71$. Although question dependence exists within certain parts of the questionnaire, this figure provides a rough estimate of just how large this probability is. The most frequently advocated method of reducing this inflated Type I error probability is through the Bonferroni correction method [Ottenbacher, 1991]. This simple procedure involves dividing the α level desired for statistical significance (in this case $\alpha = 0.05$) by the number of statistical tests conducted. Thus, through application of this method, a significant relationship will be achieved only if the ρ value is less than $\alpha = 0.0021$. As a consequence, statistical significance only remains for position and language familiarity; the other relationships indicated as statistically significant in Table 8.6 with a † do not achieve the required ρ value to be classed as significant under the Bonferroni correction method.

To conclude, although it is likely that neither medium has provided sample representativeness for this survey, the opinions expressed in the questionnaires did not

show any significant difference across the media used under the Bonferroni correction method. More importantly, the strength of the relationship between the opinion given to an arbitrary question and the medium used could only, at best, be described as moderate. Also mail based questionnaires are known to suffer from the self-selection problem within acceptable limits and these results show little difference between the two media. The problem of self-selection within the ENDQs component of this survey appears to be of a similar order to the mail distributed component, and hence there is confidence that this effect has not invalidated the results of this survey. Furthermore, this can be regarded as initial evidence suggesting that ENDQs are not fatally flawed due to the self-selection problem (see [Miller *et al.*, 1996] for a full discussion of this).

8.6.8 Positional differences

Different categories of respondents exist within the survey sample, and the data was examined for conflicting opinions between members of academia and industry. The academics, software engineers, and project managers were statistically tested for response differences. Students were not included in this analysis because of their relative inexperience; 'others' were not included because of respondent heterogeneity.

Two-tailed Chi-square tests were applied at the 0.05 α level, the results of which are presented in Table 8.7. Four significant results were obtained. First, a significant result was obtained for the capacity in which the paradigm is used: the main difference is that academics use it more for teaching purposes than for analysis and design or programming. In contrast, only a small proportion of industrialists use it for teaching.

The second significant result was obtained for the use of in-house class libraries: software engineers and project managers used these libraries significantly more often than academics. A possible explanation for this may that industrialists design for software reuse: they are producing high quality commercial applications and get a return on this investment. Academics, generally speaking, do not produce such applications and therefore do not have such a need for these libraries.

The last two significant results support the belief that academics hold a more purist view of the object-oriented paradigm. For example, not a single academics responded that it was good that C++ has become the industry de facto standard object-oriented language (see Section 8.6.6 for reasons). Furthermore, the large majority of academics were of the opinion that it was disadvantageous for object-oriented languages like C++ to allow hybrid programming. Software engineers and project managers appeared more pragmatic in their responses to these questions.

Description	Question	Degrees of freedom	Test result (X^2)	Significant?	Cramér's ϕ
Capacity used	Q. 1(b)	4	53.73	Yes	0.29
Object-oriented experience	Q. 2(a)	6	10.62	No	0.16
Language familiarity	Q. 3	4	8.47	No	0.13
Typical method size	Q. 4(a)	4	6.02	No	0.13
Method size range	Q. 4(b)	4	3.18	No	0.10
Depth of class hierarchy	Q. 5	4	3.67	No	0.10
Inheritance caused difficulty	Q. 7	6	3.46	No	0.10
Problems caused by naming	Q. 8	6	10.16	No	0.17
Use of local class libraries	Q. 9	8	18.17	Yes†	0.22
Ease of analysis and design	Q. 10(a)	2	0.20	No	0.01
Programmer productivity	Q. 10(b)	2	1.13	No	0.06
Software reuse	Q. 10(c)	2	0.99	No	0.05
Ease of maintenance	Q. 10(d)	2	0.05	No	0.01
Multiple inheritance useful	Q. 11	4	3.12	No	0.09
SP maintenance problems	Q. 12	4	4.91	No	0.12
OO maintenance problems	Q. 13	4	1.82	No	0.07
OO more maintainable than SP	Q. 14	4	4.21	No	0.11
C++ as a de facto standard	Q. 15	2	16.33	Yes	0.25
C++ allows hybrid programming	Q. 16	2	7.13	Yes†	0.16
C++ `friend` function	Q. 17	4	2.38	No	0.09
Use operator overloading	Q. 18(a)	4	2.75	No	0.10
Overload operators as	Q. 18(b)	6	9.58	No	0.20
Use of templates	Q. 19		n/a		

Table 8.7: Two-tailed Chi-square test results between each variable and the responses of the academics, software engineers and project managers. †— see text for explanation

The Bonferroni correction method was again applied to reduce the inflated probability of Type I error of test. A significant relationship will be achieved only if the ρ is less than $\alpha = 0.0022$. As a consequence, statistical significance only remains for capacity used and for C++ as a de facto standard object-oriented language; the other two relationships indicated as statistically significant in Table 8.7 with a † do not achieve the required ρ value to be classed as significant under the Bonferroni method.

Finally, the use of the Chi-square test was inappropriate for the last variable (the use of templates) because too many of the expected frequency calculations were less than five responses [Kaplan, 1987]. Examination of the dataset found no noticeable difference between academics and industrialists (the majority were less than frequent use of templates).

8.7 Validity of the survey

Questionnaire design is not an exact discipline and, as specified in Section 8.2, one of its major disadvantages is the inability to correct misunderstandings or probe responses

once the completed questionnaire has been returned. In an attempt to verify the validity of the questionnaire design, therefore, respondents' views about the questionnaire are now discussed.

A small number of respondents (3) mentioned that several questions should have had an 'it depends' category. An 'it depends' category is meaningless, however, unless respondents are willing to specify what it depends on. At the time of designing the questionnaire it was thought that this was catered for by encouraging relevant written material after each question. Such information was supplied by many respondents. In addition, two respondents commented that an invalid assumption was made about the `friend` function: it has legitimate uses as well as the illegitimate one discussed. The respondents were quite correct, but the legitimate uses of the `friend` function presented in Section 8.6.6, were not of interest in this survey. Finally, three respondents felt that the ordinal data range of Never to Always was too harsh and, consequently, respondents would be unlikely to choose the end categories. The number of responses received in these categories has shown this criticism to be inaccurate (see Appendix C).

Given the small number of criticisms levelled at the survey questions and their counter-arguments, there is justification for arguing that they had little or no impact on the validity of the survey results.

8.8 Conclusions

This chapter has considered using questionnaires as a technique for gathering empirical data and has reported the findings of an object-oriented questionnaire survey (distributed to electronic newsgroups and to a members of a mailing list) concentrating on the following issues: the perceived benefits of the object-oriented paradigm over conventional paradigms, inheritance, the understanding difficulties of object-oriented code, software maintenance and its consequences, use of local class libraries, and the C++ programming language. While there is difficulty in generalising from a sample of the population to the actual population itself, the responses of 275 object-oriented practitioners on these issues should be considered important. Results have shown that

- Respondents are of the view that the object-oriented paradigm is more advantageous than conventional paradigms in terms of ease of analysis and design, software reuse, programmer productivity, and ease of maintenance.

- Inheritance can cause difficulties when trying to understand object-oriented software: only 25% of respondents reported it had never caused them difficulty. More significant was that inheritance was catalogued as the second largest reason for understanding difficulties. It is hypothesized that understanding becomes more constrained with a deeper hierarchy (55% of respondents indicated depth of inheritance is a concept which can introduce difficulties).

- From the list of catalogued reasons for understanding difficulties, two of the first three are not paradigm specific: missing or inadequate design documentation and poor or inappropriate design and are still prevalent problems. The advantages that an improvement of current software engineering practice would bring, regardless of paradigm, should be considered.

- Maintenance is still perceived to cause software degradation, but respondents viewed (with statistical significance) this occurring less frequently *provided* the system and the change are well designed. Further, well designed object-oriented software is regarded to be more maintainable than equivalent conventional software.

- Respondents indicated that the C++ language has many deficiencies in comparison to 'purer' object-oriented languages. Consequently, the majority viewed the fact that C++ has become the industry de facto standard as detrimental.

Analysis conducted to find object-oriented language dependent answers, however, was unable to show any significant differences, one aspect of the survey findings that was contrary to expectation. Analysis was also conducted to uncover response differences between (a) respondents to the electronic questionnaire and to the mailing list questionnaire and (b) academics and industrialists. Applied statistical tests found little difference between the electronic and postal respondents: those it did find, with one exception, were neither unexpected nor inexplicable. This strengthens the findings of the survey and is also evidence that the self-selection problem of ENDQs appears to have had a minimal effect on the results of this survey. One interesting difference of opinion that arose between the academics and the industrialists was their perception of C++: academics consistently took a more purist view to object-oriented programming where industrialists appeared more pragmatic in their opinions. Finally, the validity of the survey was considered: a small minority made several criticisms which have been discussed and rejected as having any impact of the survey findings.

The questionnaire survey was phase II in the programme of research. The data collected is consistent with data collected from the interview study; it has been confirmed across a larger sample of the object-oriented practitioner population. It is concluded that the survey has been successful for gathering a large amount of quantitative data on practitioners opinions and identifying areas for further empirical investigation.

Chapter 9

Phase III: A Series Of Laboratory Experiments

9.1 Introduction

This chapter describes the third and final phase of the multi-method programme of research, subject-based laboratory experiments investigating one of the important findings demonstrated across phases I and II. A series of subject-based laboratory experiments were designed to test the effect of depth of inheritance on the maintainability of object-oriented software. In the first experiment, 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. Then an internal replication was carried out using more experienced subjects. These subjects also participated in a second experiment of similar design, but involving a greater level of inheritance depth: subjects performed identical maintenance tasks on object-oriented software with a hierarchy of five levels of inheritance depth and the equivalent object-based software. During both experiments and the replication, debriefing questionnaires were used and subjects' code was examined to assess the modification processes.

The remainder of this chapter details the design of the experiments and describes the procedures, tasks, and materials. Statistical tests are applied to the timing data collected and a detailed inductive analysis is conducted to explore alternative explanations. Conclusions are then drawn.

9.2 Experimental justification

In phase I there was consensus amongst object-oriented developers interviewed that depth of inheritance affects a programmers' ability to understand object-oriented software. In phase II, the majority of object-oriented practitioners questioned (55%) agreed that depth of inheritance is a factor when attempting to understand object-oriented software. Of these practitioners, the largest proportion indicated that between 4 and 6 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 depth of inheritance 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 [Brooks, 1980] and adopted by researchers in previous empirical studies, e.g., [Lewis *et al.*, 1992], [Porter *et al.*, 1995]. 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 [Curtis, 1986]. On the other hand, the series of experiments were conducted within a multi-method programme of research and it was hoped their results would confirm the findings of phases I and II. If confirmatory power were achieved, any conclusions drawn would be more reliable and generalisable. Subsequent studies should still seek to scale up the findings to the maintenance of more complex software by professional programmers.

9.3 Design of first experiment

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

Depth of inheritance: 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 [Schneidewind, 1987].

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 [Schneidewind, 1987].

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

> H_0 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 The use of a hierarchy of 3 levels of inheritance depth *does* affect the maintainability of object-oriented programs.

Note that no direction has been 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 phase II where difficulties begin to occur, it is not completely within that range (see Figure 8.2). 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, subjects were matched on ability and were then randomly allocated into one of two groups. Group A performed a maintenance task on a program with an inheritance hierarchy while group B performed the same 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 9.3.2 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 9.5.2).

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 [Foster, 1991]. Data gathering (discussed in Section 9.3.5) was not limited to this dependent variable to allow an inductive analysis to be performed (discussed in Section 9.5).

The design, coupled with anticipated student numbers, effectively meant that the experiment had a power level sufficient to detect any medium to large effect sizes, i.e., the experiment would have a good chance of detecting any sizable effect that exists. The power calculations for the series of experiments are presented in Appendix D.

9.3.1 Procedure

The first experiment was performed through a taught postgraduate conversion course in information technology. All of the students (see Section 9.3.2) enrolled in an object-oriented programming class using C++ which was intensively taught over a four week period with approximately nine hours of 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 on separate days 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.

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. A copy of the questionnaire is provided in Appendix D.

9.3.2 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 9.1. Allocation in this manner ensured that all subjects performed a maintenance task to both a flat program version and an inheritance

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 9.1: Group allocations to tasks in the first experiment

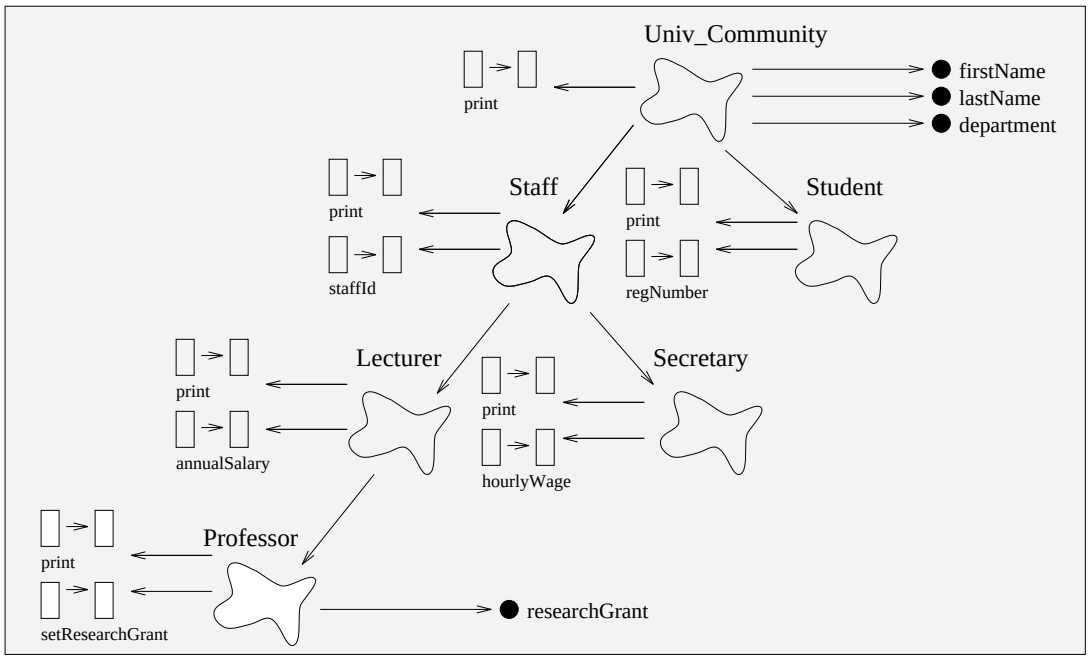


Figure 9.1: Inheritance hierarchy of database system for university staff and students.

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.

9.3.3 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 9.1 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 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 440 lines of code and consisted of four classes (each distributed in a header and implementation file). The inheritance program versions consisted of six classes and were each about 390 lines of code. 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 9.1, 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 modification task was the same.

9.3.4 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, documented source code listing of the program, 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 [Skinner, 1992].

9.3.5 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.

9.3.6 A pilot study

The importance of performing a pilot study of an experiment cannot be over emphasized. Pilot studies are conducted: (i) to find introduced assumptions in the experimental materials, (ii) to find mistakes in the experimental procedure, (iii) to test that the experimental instructions are clear, (iv) to check tasks have reasonable complexity, but that they can be completed within the allotted time, (v) to ensure performance of any automatic data collection techniques, and (vi) to attempt to identify other unforeseen circumstances. Performing a pilot study can mean the difference between a success and a failure.

Subject Identifier	Inheritance Time	Flat Time	Subject Identifier	Inheritance Time	Flat Time
1	18	44	2	39	37
3	39	31	4	-	99
5	27	-	6	45	100
7	36	64	8	-	98
9	44	-	10	48	49
11	-	-	12	38	38
13	47	67	14	25	38
15	32	26	16	36	60
17	36	-	18	-	85
19	58	78	20	57	56
21	28	35	22	36	38
23	49	29	24	52	-
25	18	41	26	92	-
27	31	46	28	41	64
29	29	-	30	102	47
31	79	-			

Table 9.2: Subjects' completion times for the first experiment (minutes)

A pilot study was performed using four experienced subjects: one for each program version. 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.

9.4 Experimental results

9.4.1 Collected time data for first experiment

Table 9.2 presents the timing data in minutes for each subject to complete the maintenance task with the inheritance program version and the flat program version for the first experiment (hyphens indicate a subject did not complete the task within the allotted time). This data is presented in summarised form in Table 9.3. 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 indicates that there were no task effects. (The difference in average times for flat versus inheritance

	$\overline{X}_{time}$	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 9.3: Statistical summary of the first experiment times

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.

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 [Brooks *et al.*, 1994]). 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 test statistic is based on the ranks of the absolute values of the difference between the two variables, giving more weight to pairs that show large differences than to pairs that show small differences. The statistical test, based upon the 20 subjects who completed both the flat and inheritance program versions, produced a significant result with $\rho = 0.05$ (two-tailed, $N = 20, z = -1.95$): 13 subjects performed better on the inheritance than the flat, six did the opposite, and one achieved the same time for both versions. Of the remaining 11 subjects who failed to complete both program versions, three subjects completed a flat but not an inheritance program version, seven subjects completed an inheritance but not a flat program version, and one subject failed to complete either program version. Together this information provides a performance ratio of 20:9, i.e., approximately two out of three subjects performed better when maintaining object-oriented software with inheritance.

9.4.2 Design of the replication and second experiment

An internal replication was conducted to confirm the direction of the findings of the first experiment, and a second experiment testing the effect of a hierarchy of five levels

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 9.4: Group allocations to tasks for the replication and second experiment

of inheritance depth was also conducted. It was decided to perform these with more experienced programmers; they were planned and executed relatively soon after the first experiment and before its results were known. Thirty one subjects,[1] a mixture of BSc. Computer Science students going into final (fourth) year and new graduates, volunteered to participate. (See Appendix D for a statistical power analysis). All subjects were well versed in C programming. The subjects participated in a taught intensive C++ course over a week, 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 9.4.3).

Subjects were allocated to two groups in the same manner as that detailed in Section 9.3.2, 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 9.4. 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 for the replication and second experiment as they were for the first experiment (see Section 9.3).

9.4.3 Collected time data for the replication and second experiment

An internal replication

For the internal replication the specified null hypothesis was the same as for the first experiment:

[1] Two subjects missed the replication because of prior commitments. All subjects participated in the second experiment.

Subject Identifier	Inheritance Time	Subject Identifier	Flat Time
3	32	1	41
4	29	2	27
5	-	6	97
7	19	10	51
8	51	14	63
9	35	15	63
11	77	16	59
12	44	18	31
13	28	20	22
17	14	22	35
19	22	23	44
21	52	25	38
24	26	27	25
26	29	29	49
28	-		

Table 9.5: Subjects' completion times for the internal replication (minutes)

H_0 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 The results of the internal replication will be in the same direction as the first experiment.

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

Table 9.5 presents the timing data in minutes for each subject to complete the maintenance task. This data is summarised in Table 9.6 in the same format used in Table 9.3. The results were in the same direction as the first experiment with a difference in time (10.9 minutes) between the two groups, 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

	$\overline{X}_{time}$	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 9.6: Statistical summary of the replication times

statistical result was not significant, $\rho = 0.07$, (one-tailed, $W = 151.0, z = -1.51$), but is arguably close enough to the 0.05 level to add some confirmatory power to the first experiment (see Section 9.5.2 for further discussion of this).

Using a deeper inheritance hierarchy

The same subjects participated in the second experiment which tested the effect of a deeper inheritance hierarchy on the maintainability of object-oriented software. The procedure used during the first experiment was kept exactly the same.

The system used for this experiment was a larger and more complex version of the university database system from the first experiment (see Figure 9.1). 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 addition, member functions were introduced so that wages and salaries could be calculated for the university employees. Figure 9.2 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 around 880 lines of code, 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 9.3.3, had 3 fewer classes (the abstract classes which were deleted), but was around 200 lines longer.

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 intended to be specialised from class Supervisor (as detailed in Figure 9.2). 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.

Standard significance testing was used with a null hypothesis of:

> H_0 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 The use of a hierarchy of 5 levels of inheritance depth *does* affect the

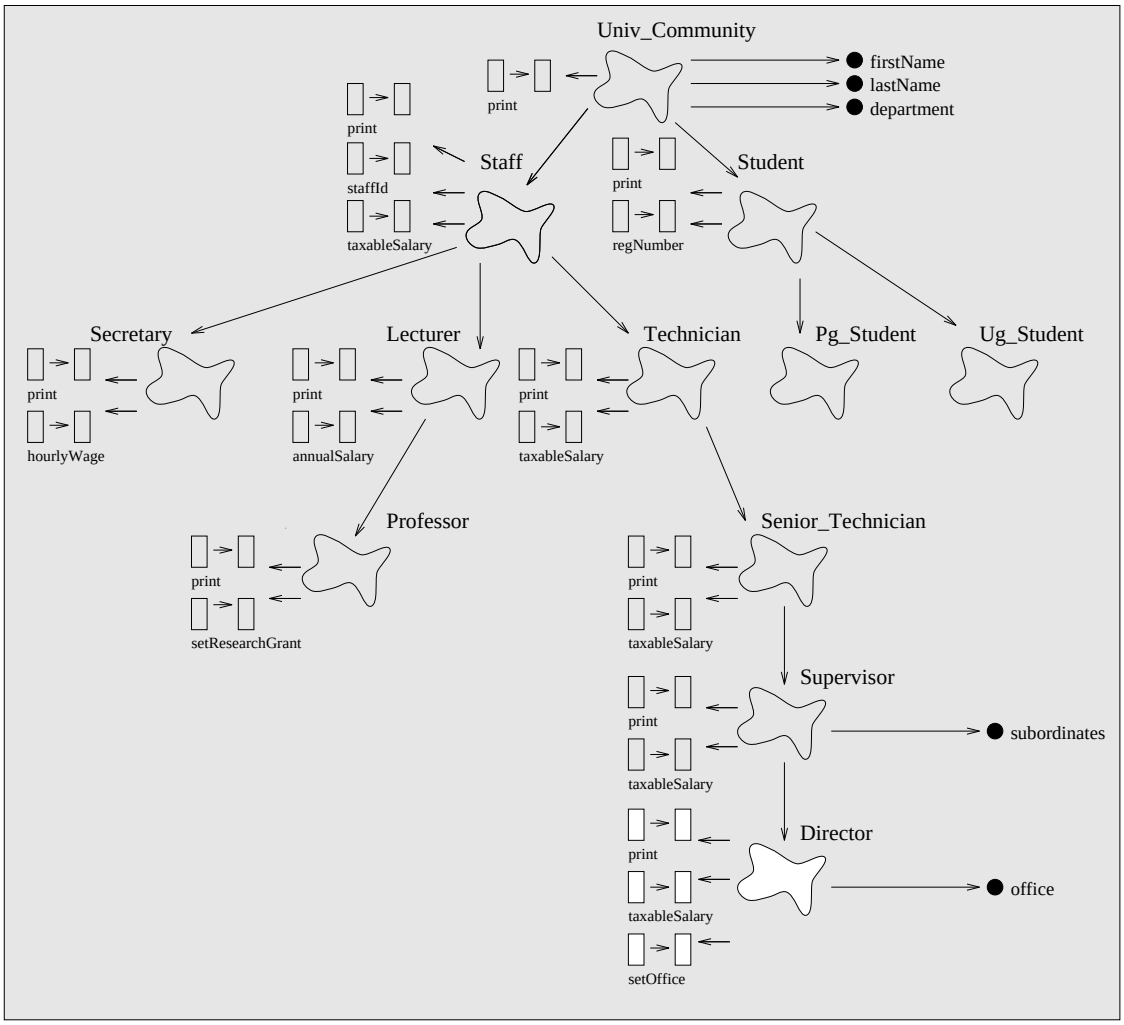


Figure 9.2: Hierarchy with 5 levels of inheritance for second experiment.

> 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.

Table 9.7 presents the timing data collected for subjects to complete the maintenance task[2]. Table 9.8 presents this timing data in the usual summarised form. Note that the direction of the mean times matches the predicted direction of the hypothesis.

[2]This includes amending the time of subject 3 from the flat group by 8 minutes due to a workstation related difficulty.

Subject Identifier	Inheritance Time	Subject Identifier	Flat Time
1	33	3	69
2	33	4	65
6	90	5	31
10	65	7	15
14	34	8	29
15	65	9	40
16	53	11	47
18	34	12	57
20	53	13	92
22	50	17	73
23	48	19	44
25	36	21	-
27	76	24	34
29	115	26	29
30	40	28	73
		31	46

Table 9.7: Subjects' completion times for the second experiment (minutes)

Cross checking the mean times from the replication and this experiment (see Tables 9.6 and 9.8) 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 $\rho = 0.27$ (one tailed, $W = 217.5, z = -0.62$), and hence the null hypothesis cannot be rejected. On the other hand, the direction of the mean times has reversed for this experiment. This is an important finding and worth exploring.

	$\overline{X}_{time}$	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 9.8: Statistical summary of the second experiment times

9.4.4 Interpretation

Figure 9.3 displays the spread of the collected times through the use of boxplots (see Appendix A or [Chambers *et al.*, 1983] 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 ($N = 15$)

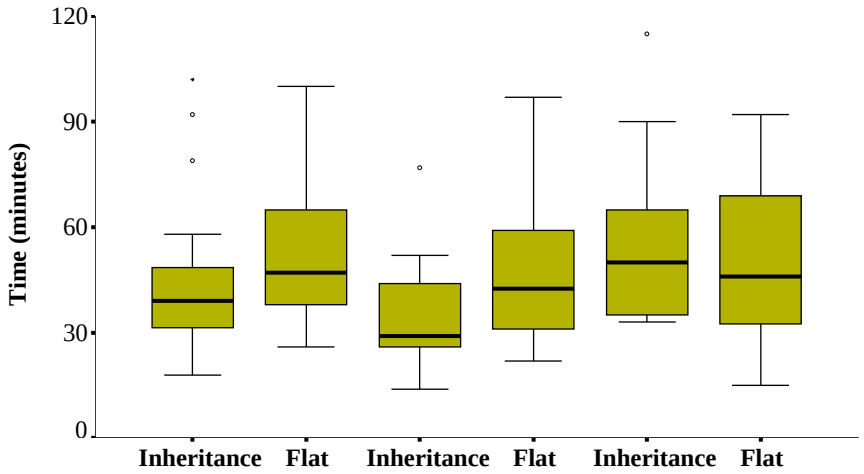


Figure 9.3: Boxplots of completion times for the first experiment, the internal replication, and the second experiment using a deeper inheritance hierarchy

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 9.4 demonstrates more clearly trends in performance. An attempt to explain the varying performances is made in Section 9.5.2.

Of immediate interest is that subjects' relative performance deteriorated on the inheritance program version with a deeper hierarchy. One possible explanation for this may be the phenomenon identified by Dvorak as conceptual entropy [Dvorak, 1994]. 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." [Dvorak, 1994].

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. In the first experiment, the majority of subjects who implemented inheritance in their solutions agreed on which class to specialise from: only four subjects, one of whom changed their mind during the modification process, did not agree with the other subjects. Further, as was expected, almost

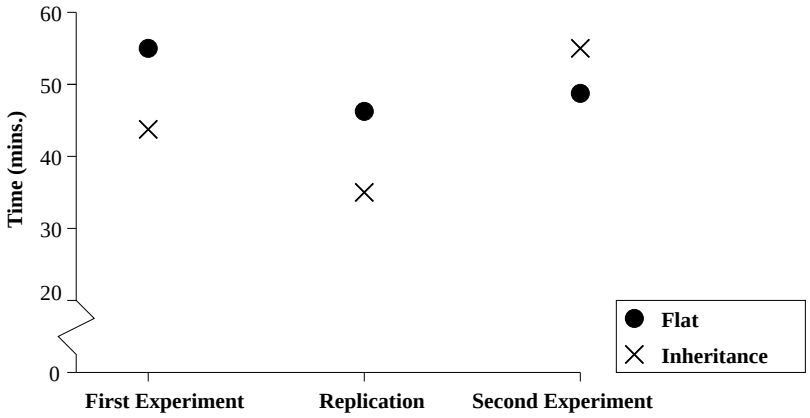


Figure 9.4: Average completion times for the first experiment, the internal replication, and the second experiment using a deeper inheritance hierarchy

all the subjects working with an inheritance program version extended its hierarchy from three to four levels of inheritance depth. Similarly, in the internal replication, only two from the 26 subjects who implemented inheritance in their solution did not specialise from the same class as the others. Again, almost all subjects working with the inheritance program version extended the hierarchy from three to four levels of inheritance depth.

In contrast, the results for the second experiment which used a deeper inheritance hierarchy found substantial disagreement about which class to specialise from amongst the subjects. Subjects working with an inheritance program version can be categorised into three groups: subjects that inherited from class Lecturer (two subjects, $\overline{X}_{time} = 57.4$ minutes), subjects that inherited from class Staff (nine subjects, $\overline{X}_{time} = 59.0$ minutes), and subjects that inherited from class Supervisor (four subjects, $\overline{X}_{time} = 47.5$ minutes). In addition, six subjects working with the flat program version attempted an inheritance solution and all of these subjects specialised from class Supervisor.

To characterise this data further, subjects' source code and debriefing questionnaires were examined. Several interesting trends emerged from subjects working with the inheritance program version. First, of the two subjects that specialised from class Lecturer, one subject thought that Lecturer was most closely related to Director. This subject had not fully understood the inheritance hierarchy, however, and this may have affected their decision to specialise from Lecturer. The other subject stated that not enough time had been spent deciding which class to specialise from; in hindsight the subject stated class Supervisor should have been chosen. Second, of the nine subjects

that specialised from class Staff, four subjects stated they had chosen it because it supported some of the functionality required. (This suggests these subjects did not look closely enough at the other classes which supported additional functionality). Four other subjects stated that class Staff was the most logical choice. (This suggests that these subjects examined the inheritance hierarchy and decided that, in reality, Director is a member of Staff, but nothing further). The remaining subject did not give a suitable reason for their choice. Third, the four subjects that chose to specialise from class Supervisor stated they did so because it had the most attributes in common with Director. (This suggests that having studied the inheritance hierarchy, these subjects decided that they should specialise from the class which most closely resembled their new class). Of the six subjects who used inheritance when working with the flat program version, four subjects stated they inherited from class Supervisor because it had the most attributes in common with Director, one subject stated Director is related to Supervisor in reality, and one subject did not give a reason for their choice. (This suggests subjects working with the flat program version were more concerned about the functionality of the superclass rather that considering if Director is a type of Supervisor in reality). These interpretations are supported by the fact that ten subjects stated in their debriefing questionnaires that deciding which class to specialise from, and the related consequences of this, had caused them the most difficulty during the experiment. In contrast, no subjects made this comment in their debriefing questionnaire for either the first experiment or the replication. It would therefore appear that extending the hierarchy from three levels of inheritance depth was less confusing than extending the hierarchy from five levels of inheritance depth.

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 [Soloway *et al.*, 1988]. According to Wilde *et al.*, the mechanism of inheritance creates further opportunities for delocalisation [Wilde *et al.*, 1993]. 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. This problem can be alleviated somewhat by a sensible naming convention which reduces the need to trace the line of method invocations in order to understand its functionality [Ponder and Bush, 1994]. 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' modification times; indeed, five subjects from the inheritance group stated in their debriefing questionnaires problems with tracing method invocations. Subjects who made these comments came from all three of the defined categories: one subject who had inherited from Lecturer, two subjects who had inherited from class Staff, and the remaining two subjects who had inherited from class Supervisor. Again, no subjects made this comment in their debriefing questionnaire for the first experiment or for the replication.

To summarise, two explanations have been uncovered as to why subjects' performance was not better on the inheritance program version with a deeper hierarchy.

1. Subjects working with a deeper inheritance program version had difficulties deciding which class to specialise from.

2. Subjects working with a deeper inheritance program version had difficulties tracing method invocations.

9.5 Inductive analysis

The concept of an inductive analysis was introduced in Chapter 2 Section 2.3.3. In this section the findings of an inductive analysis are presented. Subjects' solutions, script files, backup files, and debriefing questionnaires were all examined in an attempt to find alternative interpretations of the data.

9.5.1 Induction database

Table 9.9 displays the variables which were analysed during the inductive analysis.[3] A more detailed description of these variables, where required, is now given. Variable (1) SubjectId has been introduced to uniquely identify any given subject. Variable (3) ExpResult is the recorded time (minutes) for the subject to complete the task. Variable (4) is the time (minutes) which should be deducted from variable (3) because of either (i) adverse circumstances or (ii) inspection of subjects' program backups identified reasons for doing so, e.g., beginning a solution using inheritance, but then changing to a flat solution. Variables (9) to (20) are concerned with what the subjects wrote in the debriefing questionnaire (the grading of similar responses into groups for logical variables is discussed in Appendix D). Variable (11) ModDiff is the subjective

[3] There is slight variability between the experiments and replication for the logical value ranges provided in the table. This is because of the different response ranges given to the debriefing questionnaire.

Variable	Description	Logical values
(1) SubjectId	The subject identifier	1..31
(2) Group	The group the subject was allocated to	A or B
(3) ExpResult	The time to complete the modification task	
(4) TimeAdj	Time adjustments made to ExpResult	
(5) ExpInh	Inheritance used in modification solution	y(es),n(o)
(6) NumbComp	The number of compilations made	
(7) Experience	Programming experience	1..3
(8) Questionnaire	Completed the debriefing questionnaire	y(es),n(o)
(9) InstTime	Time to understand the experimental instructions	1..3
(10) SynDiff	Difficulty with C++ syntax	1..3
(11) ModDiff	Modification difficulty	1..10
(12) Diff	What caused the most difficulty	1..4
(13) Consume	Most time consuming action	1..3
(14) Approach	Approach to modification	a,b,c,d
(15) Understand	How well was the code understood	1..3
(16) NotUndst	Any code not understood	1..3
(17) CodeQual	Quality of code written	1..3
(18) Different	Do anything different next time	1..5
(19) Learned	Learned anything	1..4
(20) Extra	Extra comments made	1..4

Table 9.9: Induction variables

grading given by the subject for the modification task difficulty (1 - very easy, 10 - very difficult). And variable (14) Approach is the approach the subject adopted towards the modification: (a) understanding the code first, then tackling the task, (b) tackle task immediately, and attempt to understand the code as required, (c) cutting and pasting the existing files to meet the required specification, and (d) a different approach specified by subject.

Initially four separate databases for these variables were created, one each for the first and second runs of the first experiment, one for the internal replication, and one for the second experiment using a deeper inheritance hierarchy. These four databases (Tables D.1, D.2, D.3, and D.4) are presented in Appendix D along with a full description of the grading of the data into groups for the logical variables.

9.5.2 Results

The inductive analysis package IRIS [Arisholm, 1987] was used to produce a rule for each variable within each induction database. While nothing of specific value was uncovered by the induction package, conducting the inductive analysis did help to highlight several interesting patterns:

1. A weak relationship between subjects' use of inheritance and the program version (either inheritance or flat).

2. Weak evidence of a learning effect between experimental runs.

3. A case for adjustment of several subjects' completion times.

Subsequently, further analysis took place to determine if any of these patterns had an effect on the statistical analysis.

Use of inheritance within solutions to modification tasks

Data for variable (5) revealed that while all subjects used inheritance for the inheritance program versions, many subjects had also implemented inheritance for the flat program versions. This is not unnatural because subjects made extensive use of inheritance throughout their object-oriented programming class. Table 9.10 presents the number of subjects who used inheritance for their solution when working on a flat program version for comparison with those that did not (the number in parentheses is the number of subjects who did not complete the maintenance task). Two issues should be explored from these figures:

1. The different number of subjects adopting inheritance and flat solutions across the series of experiments.

2. Performance differences between subjects using inheritance in their solution and those not.

First, almost three out of five subjects produced a flat solution in the `first experiment`, but in the `replication` using more experienced subjects only about one in four did so. In the `second experiment` the number rose to three out of four subjects. Why is this? A question in the debriefing questionnaire asked subjects (a) if they had used inheritance or not in their solution and (b) to explain their reasons. Examination of this questionnaire data helped to provide the necessary insights: the majority of subjects from the `first experiment` who provided a flat solution explained that they wanted to maintain consistency between their solution and the provided flat code. In contrast, subjects from the `replication` were more eager to use inheritance because they felt (i) the class to be added was a natural specialisation of an existing class, (ii) it helped to reuse the existing code, saving time, and (iii) to reduce code redundancy. Although several subjects from the `first experiment` also made these points, it is argued that the greater experience of the `replication` subjects is the reason the majority used inheritance for the flat program version.

	Number of subjects		
Solution	First Experiment	Replication	Second Experiment
Flat	18 (3)	3 (0)	12 (0)
Inheritance	13 (5)	11(0)	4 (1)

Table 9.10: The number of subjects who used inheritance in their solution to flat program versions and the number of subjects who did not

In contrast again, it appears that subjects did not believe such reasons for using inheritance for the flat program version in the `second experiment`. Seven subjects stated that they did not use inheritance because it was not present in the existing code (four of these subjects did mention that they thought inheritance could have been used, but were not sure if it was appropriate); two subjects stated they did not feel familiar enough with inheritance to use it; two subjects stated that they had attempted to use inheritance but, having encountered difficulties, had changed to a flat solution (discussed further below); one subject gave no reason.

Second, was there any performance difference between subjects using inheritance in their solution to a flat program version and those who did not? Examination of times from the `first experiment` revealed the following: the eight subjects who used inheritance in their complete solution averaged 51.1 minutes, and the 15 subjects who produced a complete flat solution averaged 57.3 minutes. Comparison of these two times, however, does not take into account the fact that two more subjects failed to complete their solution to the flat program version when attempting to use inheritance. So there does not appear to be any real performance difference.

In the `internal replication` only three subjects produced a flat solution to the flat program version and their performance (mean of 46.3 minutes) was about average. Debriefing questionnaires were examined for explanations of their times. Subjects 2 and 14 specified their approach to the maintenance task, variable (14), had simply been to copy the relevant files and then modify them to meet the required specification. Additionally, subject 2 mentioned inheritance was not used in order to maintain consistency with the existing code and subject 14 stated a desire to simply obtain a correct result. Subject 29 had a different explanation: being confused by inheritance, subject 29 felt it would be quicker to perform the maintenance task without it. Examining times for three subjects, however, does not provide enough information to be sure of any performance difference.

In the `second experiment`, the majority of subjects produced a flat solution to the flat program version. Only four subjects used inheritance (mean of 63.0 minutes):

subjects 17 and 19 were of about average performance; subject 13 was the slowest of all the flat subjects, but blamed poor performance on having not read the source code listing properly and consequently missing out a needed function; subject 21 did not complete the task — before attempting the maintenance task, subject 21 began to construct an inheritance hierarchy from the flat class structure, an action which cost a large amount of time. Again, because limited subject numbers are being dealt with, it is difficult to say if performance differences exist.

So, overall there is no need to investigate time adjustments in order to compensate for this effect.

A possible learning effect

The possibility of a learning effect was mentioned in Section 9.3. Examination of the data found that in the `first experiment` 11 (out of 15) subjects attempted a flat solution in the first run while only seven (out of 16) subjects produced a flat solution in the second run. One simple explanation is that a handful of subjects, having used inheritance during the first run, felt more confident about using inheritance a second time. So there is marginal evidence of a learning effect. This trend was not, however, repeated by the more experienced subjects: only three subjects (out of 15) produced a flat solution in the `replication` while 12 subjects (out of 16) produced a flat solution in the `second experiment`, more than would have been expected if any learning effect existed (the reasons for this trend have been explained earlier). To be sure no learning effect was present, a two-way ANOVA test was calculated for the `first experiment`. This was to test for a sequence effect or an interaction effect on the time taken to complete a maintenance task, i.e., to discover if the sequence the subjects received the program version (flat first or inheritance first) had any effect on the maintenance task times. (A similar test cannot be performed for the `replication` and `second experiment` because the times collected are not directly comparable in this manner). The results are presented in Table 9.11 where sequence is the order a subject attempted the two programs and treatment is the program version the subject was working with. The results do not show significance for a sequence effect or for an interaction effect between the sequence and treatment effects (critical $F_{1,46,.95} = 4.05$). It is concluded that if any learning effect was present, it was sufficiently weak that it has had an insignificant impact on the statistical analysis.

Source	Sum Square	df	Mean Square	F	Sig of F
Sequence	82.04	1	82.04	0.17	0.68
Treatment	1594.89	1	1594.89	3.31	0.07
Sequence × Treatment	30.24	1	30.24	0.06	0.80
Within	22188.75	46	482.36		
Total	23895.92	49			

Table 9.11: Two-way ANOVA testing for sequence and interaction effects

Adjustments to subjects' times

As mentioned earlier it was discovered that a number of subjects began an inheritance solution but, having encountered difficulties, changed to a flat solution. Adjustment of these subjects' times was made to observe any effect on the statistical results produced in Section 9.4. Three methods of adjustment were considered:

1. Extreme adjustment — the entire time the subject was working on an inheritance solution should be subtracted from their total time.

2. Moderate adjustment — undoubtedly the subject was reading and understanding the code as they attempted their inheritance solution and hence only a percentage of this time, e.g., 50%, should be subtracted from their total.

3. Subject exclusion — exclude the subject's data from the analysis.

Although the second method appears the most objective, method (i) was adopted to see if the significant result withstood such a bias.

Two times from the `first experiment` were subsequently adjusted (subject 1: 44 to 31 minutes and subject 6: 100 to 55 minutes; adjusted mean = 52.7). The Wilcoxon signed ranks (related) was recalculated and significance became marginal with $\rho = 0.06$, (two-tailed, $N = 20, z = -1.85$). It is reassuring that, given the marginality of this result, making such biased adjustments had only a minor effect on the ρ value of the statistical test.

Extreme adjustments were also made to two flat times from the `second experiment` (subject 3: 69 to 53 minutes and subject 4: 65 to 40 minutes; adjusted mean = 46.9). While these adjustments favoured the alternative hypothesis, the resulting Wilcoxon rank sum (unrelated) still did not obtain significance $\rho = 0.15$, (one-tailed, $W = 208, z = -1.02$). Although still unable to reject the null hypothesis for this experiment evidence, in the form of (i) a change of performance direction favouring subjects who maintained the flat program version and (ii) comments made in the debriefing questionnaire, suggests that an effect may exist at deeper levels of inheritance.

Experiment	Subject Id.	Time (mins.)
First experiment	26	92
	30	102
	31	79
Internal replication	11	77
Second experiment	29	115

Table 9.12: Outlier data points from Figure 2

Certainly it warrants further investigation.

9.5.3 Outlier data

Figure 9.3 shows five datum points outlying the tail distributions of the boxplots. Each point, referred to as an outlier data point, is at least 1.5 times the distance between the 25th percentile and the 75th percentile from the 75th percentile, i.e., 1.5 times the size of the box from the top of box. Note well that the outlier data points all belong to subjects who were modifying an inheritance program version; to remove these data points would severely bias the results in favour of the inheritance times. In any case, the impact of these outliers on the results of the statistical tests was reduced because non-parametric tests were applied. Data for the subjects in question was examined in an attempt to explain why they took longer relative to other subjects.

As detailed in Table 9.12 there are three subjects times (from 27) which outlie the boxplot in the `first experiment`, one subject time (from 13) in the `internal replication`, and one subject time (from 15) in the `second experiment` using the deeper inheritance hierarchy.

The outlier subjects from the `first experiment` all had explanations for their longer times: (i) Subject 26 had quite a bit of difficulty dealing with syntax errors and removing them took the subject some time. In addition, the subject mentioned in the debriefing questionnaire that there was quite a steep learning curve for object-oriented development. (ii) Subject 30 made a simple error when editing a file and this error took approximately an hour to find. (iii) Subject 31 stated they took over half an hour to fully understand what was required for the maintenance task.

In the `internal replication` subject 11 stated it took them about 20 minutes to understand what was required for the maintenance task. Furthermore, the subject claimed to have had quite some difficulty removing syntax errors.

Finally, in the `second experiment` subject 29 had difficulty deciding which class to specialise from and twice had a change of mind about which class to use. In addition, the subject tried to tackle the task immediately and understand the code as required.

This approach may have increased their overall time.

9.6 Experimental validities

9.6.1 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., [Brooks, 1980]). An example of this has been presented in Chapter 4 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 9.3.2 and 9.4.2).

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 (see Section 9.5.2) 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 modification 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.

9.6.2 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 and complexity 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 [Chidamber and Kemerer, 1994]. 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 [Tiller, 1991] — 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., [Henry *et al.*, 1990], [Lewis *et al.*, 1992], [Porter *et al.*, 1995]. It is argued there is justification to conclude that because the effect of inheritance has been consistently reported across the programme of research, there is less of a threat to external validity. To fully overcome these threats a replication package is proposed

	Totals		
Data distribution	N	Skewness	Kurtosis
Experiment 1 flat	23	0.756	2.380
Experiment 1 inheritance	27	1.447*	4.825*
Replication flat	14	1.082*	3.966
Replication inheritance	13	1.137*	3.794
Experiment 2 flat	15	0.330	2.245
Experiment 2 inheritance	15	1.192*	3.698

Table 9.13: The results of the formal skewness and kurtosis tests. * — significant at 95% confidence interval

as further work in order to allow other researchers to conduct external replications using different subjects, variables, and procedures (see [Chapanis, 1988] for a detailed discussion about making generalisations).

9.6.3 Statistical validity

Throughout this study mainly non-parametric statistical tests have been used, namely the Wilcoxon signed ranks (related) and the Wilcoxon rank sum (unrelated). It is important to note that non-parametric tests have a smaller statistical power than their parametric equivalents and, as such, under similar conditions with a population of normal distribution the non-parametric test is less likely to reject the null hypothesis (assuming an effect does exist), i.e., the test has a larger chance of committing a Type II error. This probability was reflected by the results of the *t-test* applications whose results were slightly more significant[4]. An underlying assumption for parametric statistical tests, however, is that the sample data are drawn from a normally distributed population [Welkowitz *et al.*, 1976]. There was uncertainty if this assumption had been met — formal skewness and kurtosis tests were applied to each of the data distributions. As shown in Table 9.13 the results found several of the data distributions to be non-normal at the 95% confidence interval or better (represented by an * in the table). Although the *t-test* is a robust test and is often used when the underlying mathematical assumptions are not met, the skewness and kurtosis results mean that the non-parametric tests are more appropriate for this statistical analysis.

9.7 Conclusions

This empirical study should be of interest to those designing and maintaining object-oriented software: its results suggest that object-oriented software with a shallow

[4]The *t-test* is the parametric equivalent of the Wilcoxon tests.

hierarchy (3 levels of inheritance depth) may be more maintainable than equivalent object-based software with no inheritance. In contrast, object-oriented software with a deeper hierarchy (5 levels of inheritance depth) was not shown to be more maintainable than the equivalent object-based software; indeed, subjects' completion times, source code solutions, and debriefing questionnaires actually provided some evidence suggesting subjects began to experience difficulties with this deeper hierarchy (although no statistical significance was obtained).

While several threats to the external validity of the empirical study have been identified, it is argued that because the results have been confirmed across phases I and II of the 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 the other categories of maintenance with a view to evaluating them in terms of the overall maintenance process, not just in terms of the implementation phase.

A complete inductive analysis has been conducted and has not offered any alternative explanations of the data. The analysis considered the variability within the collected data including the use of inheritance in a flat program version, a maturation or learning effect, adjustment of subjects' times, and examination of all outlier data points. Of particular interest was the discovery of conceptual entropy — as expected the large majority of subjects extended the hierarchy from 3 to 4 levels of inheritance depth, but only a small number of subjects extended the hierarchy from 5 to 6 levels of inheritance depth. Several possible reasons for this exist, but one explanation may be that it is more demanding to extend a deeper hierarchy than it is to extend a shallower hierarchy. Schenberger has recently obtained similar results in distributed computer environments [Schneberger, 1995]. As a system is distributed into smaller and more autonomous components, Schenberger found that while the complexity of the systems components is reduced, overall complexity is increased because of an increased number of system variables; as a result, software maintenance becomes more difficult. Effectively the same occurs in an inheritance hierarchy: the deeper the hierarchy becomes, the larger the number of superclasses a class may have, and the more complex it becomes to understand their combined functionality.

Chapter 10

Evaluation

10.1 Introduction

Part III of the thesis has introduced the multi-method approach to performing empirical software engineering research. To thoroughly evaluate the multi-method approach, an application has been presented in the form of a three phased programme of research within the object-oriented paradigm. The programme of research was initiated by an empirical study involving structured interviews of experienced object-oriented developers on their opinions of the advantages and disadvantages of the object-oriented paradigm. The findings of this primary investigation were then refined and used to design and implement a questionnaire survey which was answered by 275 object-oriented practitioners. Finally, a series of subject-based laboratory experiments were conducted which investigated one of the important and most interesting findings from the questionnaire survey and structured interviews in a controlled setting.

The final chapter in this thesis part reiterates the arguments for applying the multi-method approach, summarises the results from the application of the multi-method approach, evaluates the successes and shortcomings of this application, and concludes with a series of recommendations for other researchers attempting similar work.

10.2 Justification for the multi-method approach

To achieve reliable and more generalisable results, an empirical study requires confirmatory power, usually achieved by means of: (i) external replication and (ii) investigation of the same phenomenon through a different empirical study. The most

effective method of achieving this second goal is through an approach which views a series of different empirical studies as evolutionary. That is, the important issues discovered by an initial study are refined and investigated further by the next study, and so forth. (As demonstrated by Chapters 7 through 9). The results from each study may, therefore, turn out to confirm one another.

Justification for adopting a multi-method approach to performing empirical research can be provided by three additional strengths:

1. Hypothesis formulation is no longer dependent on the experimenter's intuition. The nature of the evolutionary approach allows several hypotheses to be investigated further from the previous phase of the programme of research (which begins with an initial exploratory study). This refinement may continue until only a single hypothesis is being investigated — other less important hypotheses can be investigated later.

2. If results are consistently demonstrated across phases, conclusions drawn at the end of the programme of research are more robust. They are less likely to be adversely affected by the fact that one of the phases was poorly designed, used a biased sample of the population, misapplied statistical tests, or did not take into account one of a number of other important factors.

3. As each phase of the programme of research becomes more focused, the experimenter is likely to gain increased understanding of the various factors that affect the phenomenon under investigation. In addition, a researcher should undertake detailed investigation to determine potential causes of any inconsistencies which arise across phases. Any explanations may also provide increased understanding about the phenomenon under investigation.

As a consequence, results emerging from a multi-method programme of research are more impressive than those from a single empirical study. In turn, the software engineering community is likely to have more confidence in their reliability.

10.3 Summary of the empirical results

Applying the multi-method approach to an investigation within the object-oriented paradigm provided an interesting and important set of empirical results which have shown consistency across the first two phases of the programme of research. First, findings such as object-oriented design can be poor or inappropriate, excessive use

of inheritance can cause understanding difficulties, and a less than positive view of the most popular object-oriented language (C++) provide some evidence that object-oriented techniques are not the panacea they have been promised to be. Second, initial evidence has been provided which supports some of the purported benefits of object-oriented techniques namely, ease of analysis and design, increased programmer productivity, increased software reuse, and ease of maintenance. It should be noted that these benefits appear to be dependent on whether the problem is amenable to an object-oriented solution and the object-oriented design is appropriate. If these criteria are not met then the reverse may occur. Third, while there is some evidence to suggest that ease of maintenance is an attribute of object-oriented software, some factors arose from the programme of research which compounds this notion. Missing design documentation and poor or inappropriate design appear to be prevalent problems; these affect the understandability and maintenance of an object-oriented software system at least as much as other systems designed using alternative structured methodologies. Furthermore, the series of subject-based experiments found that depth of inheritance has an effect on the maintainability of object-oriented software. These results have implications for current software developers using object-oriented techniques and for the training of future practitioners.

10.4 Successes and shortcomings of the approach

While applying the multi-method approach to the object-oriented paradigm has produced some interesting and important findings with respect to object-oriented technology, of greater importance is that the application has provided an opportunity to evaluate the successes and shortcomings of the approach as an empirical methodology:

1. The multi-method approach has demonstrated several findings consistently across phases I and II of the programme of research and one finding consistently across all three phases, i.e., the multi-method approach has provided confirmatory power for its results, and, hence, it results are more reliable.

2. Three different cross-sections of people have been involved in the three phases of the programme of research. As a consequence of this and the confirmatory power provided by phases I and II, there is some justification for generalising the results of phase III, the series of laboratory experiments, to actual software engineering practice. If phase III has been conducted separately there would have been little justification for generalisation.

3. The multi-method approach has demonstrated that several interesting findings can arise from the more general investigations that initiate the programme of research. For example, there are findings from phases I and II of the programme of research that are worth turning into hypotheses and investigating further. The multi-method approach has also demonstrated that it is much easier for the researcher to identify the hypotheses of most importance for further investigation within the programme of research. For example, if phases I and II had not been conducted it is unlikely that the series of experiments would have investigated the effect of depth of inheritance on software maintainability. As a result of these benefits, the programme of research tends to focus on the larger and more important problems; as such, the researcher is less likely to embark on an expensive study which is of little interest or consequence.

4. Increased understanding of the various factors that affect the phenomenon under investigation was discussed as a theoretical advantage in Section 10.2. The application of the multi-method approach found that in practice this was indeed an advantage. For example, by completion of phase III much more was understood about inheritance and how depth of inheritance may affect the understandability of object-oriented software. (The inductive analysis was also an important factor in this understanding).

As with any original research undertaking there were also shortcomings:

1. The amount of time and effort required to plan, design, and organise each phase of the programme of research and analyse the collected data was considerably more than expected. In addition to the human component, i.e., the time taken to organise subject interviews, receive all completed questionnaires from respondents, and recruit subjects for the experiments, reasons for this under estimation include: (i) the effort required to analyse the verbal data collected from the structured interviews, (ii) the time taken to examine the questionnaire data for correlations between different variables, and (iii) the time and effort required to carefully plan, design, and organise (a) the series of laboratory experiments and (b) the lectures and practicals to teach the subjects the necessary object-oriented concepts.[1] In spite of this, the findings achieved from the invested time and effort to apply the multi-method approach are likely to be more cost effective than several independent studies.

[1]This was by no means an individual effort. Due acknowledgment is given to Drs. Brooks, Miller, Roper, and Wood whose efforts to achieve this goal matched, at the very least, my own.

2. The second shortcoming is a direct consequence of the first. Because the time to complete each phase of the programme of research was longer than estimated, analysis of each phase was not fully complete before the beginning of following phase. Due to constraints imposed by availability of student subjects there was little choice except to proceed to the next phase with as much of the analysis complete as possible. (This was more of an issue when moving from phase II to III than it was for moving from phase I to II). It is likely that this shortcoming is more a consequence of PhD time pressures than a shortcoming of the multi-method approach.

Although it is argued that the successes of the multi-method approach far outweigh these shortcomings, there are lessons to be learned from both.

10.5 Lessons learned

Applying the multi-method approach to an empirical investigation within the object-oriented paradigm has lead to a number of important lessons being learned. These lessons are described as a series of recommendations which should aid researchers attempting similar programmes of research.

1. The multi-method approach has proven to be a successful methodology for conducting empirical software engineering research. Its primary shortcoming, however, is the large investment of time and effort required to conduct a programme of research which involves three or more phases.
   `Recommendation:` be prepared for the time and effort required to undertake the multi-method approach for empirical work; working to deadlines can be difficult.

2. Proceeding from one phase of the multi-method approach to the next before its analysis is fully complete could lead to an investigation of lesser importance.
   `Recommendation:` rigidity should be applied when moving from one phase of the multi-method approach to the next — only when the analysis of a phase is complete should the experimenter consider the next phase.

3. It is important that each phase is recognised as an empirical study in its own right and not just part of a multi-method approach. This offers two advantages: (i) the details of each phase become available as the multi-method approach evolves, and (ii) each phase should have sufficient details reported that it can be

externally replicated should the findings be of sufficient importance to warrant it.

`Recommendation:` after the completion of the data analysis for a particular phase, the results should be reported within a technical report.

4. Results are likely to be more highly regarded when coming from a progressive programme of research like an evolutionary multi-method approach, e.g., [Daly *et al.*, 1996a], [Daly *et al.*, 1996b].

   `Recommendation:` Explicitly state which phase of the multi-method approach is being reported, how this relates to any previous phases, and what future research is planned from this work.

10.6 Additional advice

Advice, based on the experience of conducting a three phased multi-method programme of research, is also given to help researchers considering applying the approach for the first time. It should be noted that there is an element of subjectivity to this advice and it should be treated accordingly.

Phase I — the aim of the exploratory phase is to identify key issues and to explore opinions on the advantages and disadvantages of the technology under investigation. Any previous research in the area should be reviewed and, if relevant, used to help focus the phase. Phase I should use techniques which enable qualitative data to be collected. It is important that the subjects used are not particularly homogeneous as this may bias the findings and have an adverse affect on what is investigated in later phases. Phase II can be entered when the researcher is confident that enough key issues have been identified which warrant further investigation. Appropriate phase I techniques include structured interviews, a questionnaire with general open questions, verbal protocol analysis, or even a combination of these.

Phase II — the aim of phase II is to provide reliable empirical evidence from which hypotheses can be formulated for detailed investigation in the next phase. Phase II should incorporate what has been learned from phase I and investigate it further by collecting a sizable amount of quantitative data. The use of inferential statistics can help identify large effects — these can be rated in order of interest

to the software engineering community. Phase III can be entered if effects of sufficient importance and interest are found and can be turned into hypotheses for more detailed investigation. Appropriate phase II techniques include widely distributed questionnaires, more general case studies, and verbal and video protocol analysis.

Phase III — the aim of phase III is to provide strong empirical support for the one or more hypotheses under investigation. Three phases is the minimum number of phases recommended for the multi-method approach. Phase III may be an exit point (i.e., enough data has been collected) if the findings have been consistently demonstrated by this phase and earlier phases of the programme of research. Appropriate techniques for phase III (if an exit point) are laboratory experiments and detailed case studies.

Further phases — depending on the techniques used in earlier phases further investigation using, e.g., professional programmers or industrial scale software, may be required to allow any generalisations.

10.7 Conclusions

The multi-method approach allows focused empirical enquiry to be undertaken from a more general empirical study. Moreover, if an effect has been consistently demonstrated then this approach adds confirmatory power; subsequently, the software engineering community are more likely to have confidence in the reliability of the findings. Similarly, if different population samples are used for different phases of the programme of research and results are consistently demonstrated, there is some justification for generalising the results.

Although the multi-method approach requires a large investment of time and effort to apply it to a programme of empirical research, it has been argued that the return on this investment provides several other advantages including, ease of identification of hypotheses for further investigation, focusing on the more important problems as the programme of research progresses, increased understanding of the various factors that affect the phenomenon under investigation, and there is less chance of conducting an expensive study which is of little value or consequence. It is concluded that researchers should consider applying a multi-method approach paying close attention to the recommendations that have been made above.

Part IV

CONCLUSIONS

Chapter 11

Conclusions And Further Work

11.1 Summary of thesis

Increasingly large amounts of money are being consumed by software costs, yet problems still exist with software quality and delivering software on schedule and within the development budget. This is partly as a result of the absence of measurement programmes and partly as a result of software developers using software technology which has not been evaluated. As a consequence, conducting empirical evaluation has started to become a more important part of software engineering research. The methodology used for empirical software engineering research is still somewhat immature, however, and requires improvement for several reasons. First, much of the empirical research that has been conducted contains weaknesses as a result of the empirical methodology used. Second, empirical results are not being confirmed before being accepted by the software engineering community. And, third, much of the empirical software engineering research has not included analysis of the data for alternative interpretations. To understand software development better and manage it more cost-effectively more methodical empirical research is required. This thesis is concerned with achieving this goal by means of a research methodology using the multi-method approach integrated with the technique of replication.

The research methodology has been used to present (i) the results of an external replication of a well performed software engineering empirical study and (ii) an application of the multi-method approach to an investigation within the object-oriented

paradigm. Conducting the external replication study and the multi-method programme of research has demonstrated why these two techniques are able to alleviate some of the problems surrounding current empirical software engineering research methodology.

11.2 Thesis results and further work

11.2.1 Results

Part II of this thesis details the experiences of conducting an external replication of a well performed empirical study. Although the results of the external replication did not repeat the results of the original study, conducting an inductive analysis helped to understand the reasons for the difference. The most important lessons learned from conducting the external replication are:

1. Careful consideration of the purpose of conducting an external replication study. If the purpose is to confirm the original results then only minor recipe-improvements should be made. If the purpose is to attempt to generalise the results then major recipe-improvements or alternatives must be made. (Note well: if improvements or alternatives are too substantial, it becomes debatable whether the study counts as an external replication).

2. The level of reported detail should be sufficient to allow other researchers to attempt an external replication (or, if not, the detail should be available on request). The production of a laboratory kit can substantially reduce the amount of work required by researchers to undertake an external replication. It is important that researchers also begin to report an estimate of the size of the effect they are investigating. This will allow researchers undertaking an external replication to conduct a statistical power analysis thus ensuring the replication has a sufficient level of statistical power.

3. The relative ability of the subjects is likely to be one of the largest sources of variability in an empirical study involving humans. Researchers conducting such research should carefully consider the ability of the subjects and attempt to control for any effect it could have on the results in the experimental design.

As a consequence of conducting the external replication study, a framework has been established for categorising external replications of other software engineering empirical studies as they take place (see Chapter 3, Section 3.3).

Part III of this thesis details the multi-method approach and the results of an application within the object-oriented paradigm. Conducting a multi-method programme of research allowed the strengths and weaknesses of the approach to be evaluated. Realised strengths include: (i) Hypothesis formulation is no longer dependent on the researcher's intuition — by conducting an exploratory survey gathering qualitative data, a number of important findings can be turned into hypotheses for further investigation, i.e., the researcher is likely to become aware of issues they had not previously considered. (ii) As hypotheses become more focused for each phase of the programme of research, increased understanding of factors which affect the phenomenon being investigated is gained. (iii) If results can be consistently demonstrated across phases of the programme of research more confidence can be placed in the reliability of the findings. (iv) Conclusions emerging from a programme of research are robust in that they are less likely to be affected if one of the phases has weaknesses identified in its design. If weaknesses are identified in the design of a single empirical it is likely the findings become regarded as invalid.

The main weakness of the approach is the large investment of time and effort required to conduct a multi-method programme of research. This investment includes carefully planning and controlling the programme of research to enable analysis of each phase prior to beginning the next phase. Nevertheless, a multi-method programme of research is likely to be more cost effective than several independent studies because more reliable and generalisable conclusions can be achieved.

The most important lessons learned from applying the multi-method approach are:

1. Results emerging from a multi-method programme of research are likely to be more highly regarded by the software engineering community.

2. Each phase of the programme of research should be treated as an independent empirical study and ideally written up as technical report in its own right. This means the details of the programme of research become available as each phase is completed rather than when the whole programme of research is completed. More importantly, it should mean that sufficient detail is reported for each phase should researchers be interested in conducting an external replication.

11.2.2 Further work

As a result of conducting the multi-method programme of research, three main areas of further work are identified:

- The application of the multi-method approach within the object-oriented paradigm has discovered a set of interesting results. In particular, there are several findings consistently demonstrated across phases I and II of the programme of research that could be stated as hypotheses for further empirical investigation. For example, phases I and II found that:
  - The design of object-oriented software is considered to be more important than the design of conventional software. It would be interesting to test if poorly designed object-oriented software is more difficult to understand and maintain than poorly designed structured software.
  - Design documentation is perceived to be a very important aid to understanding object-oriented software. It may be that without design documentation object-oriented software is no easier or, indeed, more difficult to understand that undocumented conventional software. In contrast, if design documentation is available object-oriented software may be easier to understand. There is also scope to incorporate design documentation into the series of experiments that were conducted on the effect of inheritance depth — design documentation may alleviate some of the difficulties encountered.
  - Object-oriented software is considered to resist degradation from maintenance under the condition that changes made are appropriately designed. It would be interesting to test the effect of making such changes to object-oriented software and contrast them with the effect of making 'quick fixes'.
- The series of subject-based laboratory experiments could be made available as a package for external replication should any researcher be interested in verifying the results. Such a package would require that all source code, experimental instructions, and data collection facilities are included as well as the information provided within this thesis.
- To fully generalise the results of the series of subject-based experiments there should be some attempt to scale up the findings to professional programmers using more complex software.

11.3 Conclusions

It has been argued that the methodology for conducting empirical software engineering research requires improvement. This thesis provides a number of contributions to an improved methodology:

1. It has been shown that replication and the multi-method approach are key elements of confirmatory research. External replication seeks to provides confidence in the results of an original study whereas the multi-method approach provides confirmatory power within a programme of empirical research. As a result, findings are likely to be more reliable and generalisable.

2. A multi-method approach helps to address many of the weaknesses that exist in current empirical research. For example, the approach facilitates investigation of more important hypotheses, more thorough data analysis, more methodical reporting, and greater understanding of the factors that affect the phenomenon under investigation. Furthermore, the multi-method approach ensures a greater coverage of the problem space through the exploratory phases of the programme of research. This may encourage researchers to undertake more empirical research and thus address the lack of empirical results within software engineering.

3. The inductive analysis paradigm has been used to ensure thorough analysis of the collected data and to ensure any alternative interpretations of the data have been considered.

In conclusion, researchers should adopt a methodology for empirical software engineering research which integrates the multi-method approach with the technique of replication. Such a methodology will, over time, help to provide more reliable and generalisable empirical results within software engineering. In addition, when dealing with human subjects, inductive analysis should be conducted to ensure alternative interpretations are not overlooked.

Bibliography

[Amir and Sharon, 1991] Y. Amir and I. Sharon. Replication research: A "must" for the scientific advancement of psychology. In J. Neuliep, editor, *Replication in the Social Sciences*, pages 51–69. Sage Publications, first edition, 1991.

[Arisholm, 1987] G. Arisholm. IRIS — Integrated Rule Induction System. *Department of Computer Science, University of Strathclyde, Glasgow*, 1987.

[Bainbridge, 1990] L. Bainbridge. Verbal protocol analysis. In J. Wilson and E. Corlett, editors, *Evaluation of Human Work, A practical ergonomics methodology*, pages 161–179. Taylor and Francis, 1990.

[Banker *et al.*, 1993] R. Banker, S. Datar, C. Kemerer, and D. Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–94, 1993.

[Baroudi and Orlikowski, 1989] J. Baroudi and W. Orlikowski. The problem of statistical power in MIS research. *MIS Quarterly*, 13:87–106, March 1989.

[Basili and Reiter, 1981] V. Basili and R. Reiter. A controlled experiment quantitatively comparing software development approaches. *IEEE Transactions on Software Engineering*, SE-7(3):299–313, 1981.

[Basili and Rombach, 1988] V. Basili and H. Rombach. The TAME project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, SE-10(6):758–773, 1988.

[Basili and Selby, 1987] V. Basili and R. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, SE-13(12):299–313, 1987.

[Basili and Selby, 1991] V. Basili and R. Selby. Paradigms for experimentation and empirical studies in software engineering. *Reliability Engineering and System Safety*, 32:171–191, 1991.

[Basili and Weiss, 1984] V. Basili and D. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, SE-10(6):728–738, 1984.

[Basili *et al.*, 1986] V. Basili, R. Selby, and D. Hutchens. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, SE-12(7):733–743, 1986.

[Basili, 1992] V. Basili. The experimental paradigm in software engineering. In H. Rombach, V. Basili, and R. Selby, editors, *Experimental Software Engineering Issues: Critical Assessment and Future Directions*, Lecture Notes in Computer Science 706, pages 3–12. Springer-Verlag, 1992.

[Berard, 1993] E. Berard. *Essays On Object-Oriented Software Engineering*, volume 1. Prentice Hall, 1993.

[Biemer *et al.*, 1991] P. Biemer, R. Groves, L. Lyberg, N. Mathiowetz, and S. Sudman, editors. *Measurement Errors in Surveys*. John Wiley and Sons, Inc, 1991.

[Booch, 1986] G. Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, SE-12(2):211–221, February 1986.

[Brewer and Hunter, 1989] J. Brewer and A. Hunter. *Multimethod Research A Synthesis of Styles*. Sage Publications, 1989.

[Broad and Wade, 1986] W. Broad and N. Wade. *Betrayers of the Truth*. Oxford University Press, 1986.

[Brooks and Vezza, 1989] A. Brooks and P. Vezza. Inductive analysis applied to the evaluation of a CAL tutorial. *Interacting with Computers, The Interdisciplinary Journal of Human Computer Interaction 1*, pages 159–170, 1989.

[Brooks *et al.*, 1987] A. Brooks, A. Walker, and C. Boardman. At the interface of shell built expert systems. *Third International Expert Systems Conference*, June 1987.

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

[Brooks *et al.*, 1995] A. Brooks, J. Daly, J. Miller, M. Roper, and M. Wood. Replication of experimental results in software engineering. Research report EFoCS-17-94, Department of Computer Science, University of Strathclyde, Glasgow, 1995.

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

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

[Buzzard and Mudge, 1985] G. Buzzard and T. Mudge. Object-based computing and the Ada programming language. *IEEE Computer*, 18(3):12, 1985.

[Card *et al.*, 1986] D. Card, V. Church, and W. Agresti. An empirical study of software design practices. *IEEE Transactions on Software Engineering*, SE-12(2):264–271, February 1986.

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

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

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

[Cohen, 1969] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences.* Academic Press, first edition, 1969.

[Collins, 1985] H. Collins. *CHANGING ORDER Replication and Induction in Scientific Practice.* SAGE Publications, 1985.

[Coolican, 1990] H. Coolican. *Research Methods and Statistics in Psychology.* Hodder & Stoughton, 1990.

[Courtney and Gustafson, 1993] R. Courtney and D. Gustafson. Shotgun correlations in software measures. *Software Engineering Journal*, 8(1):5–13, 1993.

[Cox, 1986] B. Cox. *Object-Oriented Programming.* Addison-Wesley, first edition, 1986.

[Crocker and von Mayrhauser, 1993] 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.

[Curtis, 1980] B. Curtis. Measurement and experimentation in software engineering. *Proceedings of the IEEE*, 68(9):1144–1157, September 1980.

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

[Daly *et al.*, 1994] 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.

[Daly *et al.*, 1995a] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. The effect of inheritance on the maintainability of object-oriented software: An empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 20–29, October 1995.

[Daly *et al.*, 1995b] J. Daly, J. Miller, A. Brooks, M. Roper, and M. Wood. A multi-method approach to performing empirical research. *IEEE Software Engineering Technical Council Newsletter*, 14(1):SPN 10–SPN 12, 1995.

[Daly *et al.*, 1995c] J. Daly, J. Miller, A. Brooks, M. Roper, and M. Wood. A survey of experiences amongst object-oriented practitioners. In *Proceedings of the IEEE Asia-Pacific Software Engineering Conference*, pages 137–146, December 1995.

[Daly *et al.*, 1996a] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. Evaluating the effect of inheritance on the maintainability of object-oriented software. In *Empirical Studies of Programmers: Sixth Workshop*, pages 39–57. Ablex Publishing Corporation, January 1996.

[Daly *et al.*, 1996b] J. Daly, J. Miller, A. Brooks M. Roper, and M. Wood. Poster presentation: An empirical evaluation of object-oriented practitioners' experiences. In *Empirical Studies of Programmers: Sixth Workshop*, pages 267–268. Ablex Publishing Corporation, January 1996.

[Davis, 1994] A. Davis. Fifteen principles of software engineering. *IEEE Software*, 11(6):94–101, November 1994.

[Denning *et al.*, 1990] S. Denning, D. Hoiem, M. Simpson, and K. Sullivan. The value of thinking aloud protocols in industry: A case study at Microsoft Corporation. In

Proceedings of the Human Factors Society 34th Annual Meeting, pages 1285–1289, 1990.

[Deubler and Koestler, 1994] H. Deubler and M. Koestler. Introducing object-orientation in large and complex systems. *IEEE Transactions on Software Engineering*, 20(11):840–848, November 1994.

[Dillman, 1978] D. Dillman. *Mail and telephone surveys: the total design method.* John Wiley & Sons, first edition, 1978.

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

[Edwards, 1972] B. Edwards. *Statistics for Business Students.* Collins, first edition, 1972.

[Ericsson and Simon, 1984] A. Ericsson and H. Simon. *Protocol Analysis.* The MIT Press, first edition, 1984.

[Fenton *et al.*, 1994] N. Fenton, S. Pfleeger, and R. Glass. Science and substance: A challenge to software engineers. *IEEE Software*, 11(4):86–95, July 1994.

[Fenton, 1994] N. Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions in Software Engineering*, 20(3):199–206, 1994.

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

[Frakes and Pole, 1994] W. Frakes and T. Pole. An empirical study of representation methods for reusable software components. *IEEE Transactions on Software Engineering*, 20(8):617–630, August 1994. Special Issue on 1993 European Software Engineering Conference.

[Gamma *et al.*, 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, first edition, 1995.

[Glass, 1994] R. Glass. The software research crisis. *IEEE Software*, 11(6):42–47, November 1994.

[Goguen and Linde, 1993] J. Goguen and C. Linde. Techniques for requirements elicitation. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 152–164, 1993.

[Goldstein and Goldstein, 1978] M. Goldstein and I. Goldstein. *HOW WE KNOW An Exploration of the Scientific Process.* Plenum Press, New York and London, 1978.

[Hart, 1985] A. Hart. Experience in the use of an inductive system in knowledge engineering. In M A Bramer, editor, *Research and Development in Expert Systems*, pages 117 – 126. Cambridge University Press, 1985.

[Hawkins, 1975] D. Hawkins. Estimation of nonresponse bias. *Sociological Methods and Research*, 3:461–485, 1975.

[Hendrick, 1991] C. Hendrick. Replications, strict replications, and conceptual replications: Are they important? In J. Neuliep, editor, *Replication in the Social Sciences*, pages 41–49. Sage Publications, first edition, 1991.

[Henricson and Nyquist, 1992] M. Henricson and E. Nyquist. *Programming in C++ Rules and Recommendations.* Ellemtel Telecommunications Systems Laboratories, 1992.

[Henry *et al.*, 1990] 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.

[Johnson and Foote, 1988] R. Johnson and B. Foote. Designing reusable software. *Journal of Object-Oriented Programming*, 1:25–35, June/July 1988.

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

[Jüttner *et al.*, 1994] 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.

[Kaplan, 1987] R. Kaplan. *Basic Statistics for the Behavioral Sciences.* Allyn and Bacon, Inc., first edition, 1987.

[Keppel *et al.*, 1992] G. Keppel, W. Saufley, and H. Tokunaga. *Introduction to Design and Analysis.* W. H. Freeman and Company, first edition, 1992.

[Kikuchi *et al.*, 1993] T. Kikuchi, T. Matsuoka, T. Takeda, and K. Kishi. Automatic classification of a large volume of questionnaire data by means of competitive learning. In *World Congress on Neural Networks*, pages I212–I215, 1993.

[Kitchenham *et al.*, 1994] B. Kitchenham, S. Linkman, and D. Law. Critical review of quantitative assessment. *Software Engineering Journal*, 9(2):43–53, 1994.

[Korson and Vaishnavi, 1986] T. Korson and V. Vaishnavi. An empirical study of the effects of modularity on program modifiability. In *Empirical Studies of Programmers: First Workshop*, pages 168–186. Ablex Publishing Corporation, 1986.

[Korson, 1986] T. Korson. *An Empirical Study of the Effects of Modularity on Program Modifiability*. PhD thesis, College of Business Administration, Georgia State University, 1986.

[Kung *et al.*, 1994] 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.

[Lamal, 1991] P. Lamal. On the importance of replication. In J. Neuliep, editor, *Replication in the Social Sciences*, pages 31–35. Sage Publications, first edition, 1991.

[Lee and Pennington, 1994] A. Lee and N. Pennington. The effects of paradigm on cognitive activities in design. *International Journal of Human-Computer Studies*, 40:577–601, 1994.

[Lejter *et al.*, 1992] 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.

[Lewis *et al.*, 1991] J. Lewis, S. Henry, D. Kafura, and R. Schulman. An empirical study of the object-oriented paradigm and software reuse. In *OOPSLA*, pages 184–196, 1991.

[Lewis *et al.*, 1992] 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.

[Liberherr and Holland, 1989] K. Liberherr and I. Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6:38–48, September 1989.

[Liberherr and Xiao, 1993] K. Liberherr and C. Xiao. Object-oriented software evolution. *IEEE Transactions on Software Engineering*, 19(4):1993, April 1993.

[Lientz and Swanson, 1980] B. Lientz and E. Swanson. *Software Maintenance Management.* Addison-Wesley, first edition, 1980.

[Lohse and Zweben, 1984] J. Lohse and S. Zweben. Experimental evaluation of software design principles: An investigation into the effect of module coupling on system modifiability. *Journal of Systems and Software*, 4:301–308, 1984.

[Lott and Rombach, 1995] C. Lott and H. Rombach. A repeatable software engineering experiment for comparing defect-detection techniques. Technical report, Department of Computer Science, University of Kaiserslautern, Germany, 1995.

[Lozinski, 1991] C. Lozinski. Why I need Objective-C. *Journal of Object-Oriented Programming*, pages 21–28, September 1991.

[MacDonell, 1991] S. MacDonell. Rigor in software complexity measurement experimentation. *Journal of Systems and Software*, 16(2):141–149, 1991.

[Mancl and Havanas, 1990] D. Mancl and W. Havanas. A study of the impact of C++ on software maintenance. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 63–69, 1990.

[Marchionini, 1990] G. Marchionini. Evaluating hypermedia-based learning. In D. Jonassen and H. Mandel, editors, *Designing hypertext/hypermedia for learning*, pages 355–373. Springer-Verlag, 1990.

[Marciniak, 1994] J. Marciniak, editor. *Encyclopedia of Software Engineering*, volume 1 and 2. John Wiley and Sons, Inc., 1994.

[McDermid, 1994] J. McDermid, editor. *Software Engineer's Reference Book.* Butterworth-Heinemann Ltd, 1994.

[McNeill, 1985] P. McNeill. *Research Methods.* Tavistock Publications, first edition, 1985.

[Miller *et al.*, 1994] J. Miller, G. Darroch, M. Wood, A. Brooks, and M. Roper. Changing programming paradigm - an empirical investigation. In M. Lee, B. Barta, and P. Juliff, editors, *Software Quality and Productivity*, pages 62–65. Chapman and Hall, 1994.

[Miller *et al.*, 1995] J. Miller, J. Daly, M. Wood, A. Brooks, and M. Roper. Statistical power and its subcomponents - missing and misunderstood concepts in software

engineering empirical research. Research report EFoCS-15-95, Department of Computer Science, University of Strathclyde, Glasgow, 1995.

[Miller *et al.*, 1996] J. Miller, J. Daly, A. Brooks, M. Roper, and M. Wood. Electronic bulletin board distributed questionnaires for exploratory research. *Journal of Information Technology*, 22(2), to appear April 1996.

[Miller, 1975] S. Miller. *Experimental Design and Statistics.* Essential Psychology. Methuen, 1975.

[Mitchell *et al.*, 1987] J. Mitchell, J. Urban, and R. McDonald. The effect of abstract data types on program development. *IEEE Computer*, pages 85–88, August 1987.

[Moher and Schneider, 1982] T. Moher and G. Schneider. Methodology and experimental research in software engineering. *International Journal of Man-Machine Studies*, 16:65–87, 1982.

[Moreau and Dominick, 1990] D. Moreau and W. Dominick. A programming environment evaluation methodology for object-oriented systems: Part II - test case application. *Journal of Object-Oriented Programming*, 3(3):23–32, 1990.

[Neale and Liebert, 1986] J. Neale and R. Liebert. *Science and Behaviour: An introduction to methods of research.* Social Learning Theory. Prentice-Hall, third edition, 1986.

[Neilson, 1992] J. Neilson. Evaluating the thinking-aloud technique for use by computer scientists. In R. Hartson, editor, *Advances in Human Computer Interaction*, volume 3, pages 69–82. Ablex Publishing Corporation, 1992.

[Oppenheim, 1992] A. Oppenheim. *Questionnaire design, interviewing, and attitude measurement.* Pinter Publishers, new edition, 1992.

[Ottenbacher, 1991] K. Ottenbacher. Statistical conclusion validity. *American Journal of Physical Medicine and Rehabilitation*, 70(6):317–322, 1991.

[Page-Jones, 1988] M. Page-Jones. *Practical Guide to Structured Systems Design.* Prentice-Hall International, second edition, 1988.

[Perry and Kaiser, 1990] D. Perry and G. Kaiser. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming*, 2(1):13–19, 1990.

[Pfleeger, 1994] S. Pfleeger. Design and analysis in software engineering. Part 1: The Language of Case Studies and Formal Experiments. *ACM SIGSOFT Software Engineering Notes*, 19(4):16–20, October 1994.

[Pfleeger, 1995] S. Pfleeger. Design and analysis in software engineering. Part 3: Types of Experimental Design. *ACM SIGSOFT Software Engineering Notes*, 20(2):14–16, April 1995.

[Pokkunuri, 1989] B. Pokkunuri. Object-oriented programming. *SIGPLAN Notices*, 24(11):96–101, 1989.

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

[Popper, 1968] K. Popper. *The Logic of Scientific Discovery.* Hutchinson of London, revised edition, 1968.

[Porter *et al.*, 1995] 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.

[Potts, 1993] C. Potts. Software engineering research revisited. *IEEE Software*, 10(5):19–28, September 1993.

[Pratto and Rodman, 1987] D. Pratto and H. Rodman. Magazine-distributed questionnaires for exploratory research: Advantages and problems. *Sociological Spectrum*, 7:61–72, 1987.

[Pressman, 1994] R. Pressman. *Software Engineering: A practitioner's approach.* McGraw Hill, European edition, 1994.

[Quinlan, 1986] J. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Robins, 1988] J. Robins. Attributions and depression: Why is the literature so inconsistent? *Journal of Personality and Social Psychology*, 54(5):880–889, 1988.

[Roper, 1992] M. Roper. Software testing: A selected annotated bibliography. *Software testing, verification and reliability*, 2:113–132, 1992.

[Rosenthal, 1991] R. Rosenthal. Replication in behavioral research. In J. Neuliep, editor, *Replication in the Social Sciences*, pages 1–30. Sage Publications, first edition, 1991.

[Rumbaugh *et al.*, 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[Sawyer and Ball, 1981] A. Sawyer and D. Ball. Statistical power and effect size in marketing research. *Journal of Marketing Research*, 18(3):275–290, August 1981.

[Scanlan, 1989] D. Scanlan. Structured flowcharts outperform pseudocode: An experimental comparison. *IEEE Software*, 6(5):28–36, September 1989.

[Schlaer and Mellor, 1988] S. Schlaer and S. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data.* Yourdon Press: Prentice Hall, first edition, 1988.

[Schneberger, 1995] 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.

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

[Sharpe *et al.*, 1991] S. Sharpe, D. Haworth, and D. Hale. Characteristics of empirical software maintenance studies: 1980-1989. *Journal of Software Maintenance: Research and Practice*, 3:1–15, 1991.

[Shneiderman *et al.*, 1977] B. Shneiderman, R. Mayer, D. McKay, and P. Heller. Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM*, 20(6):373–381, 1977.

[Sinclair, 1990] M. Sinclair. Subjective assessment. In J. Wilson and E. Corlett, editors, *Evaluation of Human Work: A practical ergonomics methodology*, pages 58–88. Taylor and Francis, 1990.

[Sinha and Vessey, 1992] A. Sinha and I. Vessey. Cognitive fit: An empirical study of recursion and iteration. *IEEE Transactions on Software Engineering*, 18(5):368–378, 1992.

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

[Smith, 1983] G. Smith. The problems of reduction and replication in the practice of the scientific method. *Annals of the New York Academy of Sciences*, 406:1–4, 1983.

[Soloway *et al.*, 1988] 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.

[Stroustrup, 1991] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.

[Sudman and Bradman, 1983] S. Sudman and N. Bradman. *Asking Questions*. Jossey-Bass Publishers, first edition, 1983.

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

[van Hillegersberg *et al.*, 1995] J. van Hillegersberg, K. Kumar, and R. Welke. An empirical analysis of the performance and strategies of programmers new to object-oriented techniques. In *Psychology of Programming Interest Group: 7th Workshop*, January 1995.

[Votta and Porter, 1995] L. Votta and A. Porter. Experimental software engineering: A report on the state of the art. In *Proceedings of the IEEE International Conference on Software Engineering*, pages 277–279, 1995.

[Walz *et al.*, 1993] D. Walz, J. Elam, and B. Curtis. Inside a software design team: Knowledge acquisition, sharing and integration. *Communications of the ACM*, 36(10):63–76, 1993.

[Welkowitz *et al.*, 1976] J. Welkowitz, R. Ewen, and J. Cohen. *Introductory Statistics for the Behavioral Sciences*. Academic Press, second edition, 1976.

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

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

[Zweben *et al.*, 1995] S. Zweben, S. Edwards, B. Weide, and J. Hollingsworth. The effects of layering and encapsulation on software development cost and quality. *IEEE Transactions on Software Engineering*, 21(3):200–208, March 1995.

Appendix A

Glossary

Definitions given in this glossary are the accepted definitions used by the International Software Engineering Research Network (ISERN) or, if not yet ISERN defined, common definitions found in any empirical or statistical text. ISERN definitions have been given priority because the network consists of a group of organisations, universities, and companies, from all over the world, who place the same importance on conducting empirical research. ISERN members are the leading software engineering institutions conducting empirical work and meet annually to exchange ideas and share knowledge.

Alternative hypothesis (H_1): the hypothesis that remains tenable when the null hypothesis is rejected.

Availability sampling: obtaining a sample of the population simply by making use of people who are available and willing to participate in an experiment.

Boxplot: used to represent the distribution of a set of data where the bottom of the box represents the 25th percentile, the top of the box the 75th percentile, and the line across the middle the 50th percentile (median value). The top hinge is the largest data point less than or equal to 1.5 midspreads (the length of the box) above the 75th percentile; the bottom hinge is the smallest point greater than or equal to 1.5 midspreads below the 25th percentile. An outlier is any point above or below a hinge.

Confirmatory power: providing confidence in the findings of an empirical study by either replicating it and achieving similar results or investigating a similar

hypothesis by means of a different empirical technique and achieving similar results.

Dependent variable: a measure used to characterise the effects of the independent variable(s).

Effect size: the degree to which the phenomenon under study is present in the population.

Empirical: said of data based on observation or experience and of findings that can be verified by observation or experience.

Experiment: in general, an experiment is defined as an act or operation for the purpose of discovering something unknown or testing a principle, supposition, etc;

In software engineering, a trial that is conducted in order to verify a hypothesis defined beforehand in a controlled setting in which the most critical factors can be controlled or monitored.

Hypothesis: a tentative explanation that accounts for a set of facts and can be tested by further investigation; a theory.

Independent variable: a manipulation that is applied to the subject of study.

Null hypothesis (H_0): a statement concerning one or more parameters that is subjected to a statistical test.

Population: all observations of the phenomena being studied, e.g., all software modules, all code reviews, all programmers.

Power of a statistical test: probability of rejecting the null hypothesis when the alternative hypothesis is true.

Qualitative data: data represented as words and pictures, not numbers. **Qualitative analysis** consists of methods designed to make sense of qualitative data.

Quantitative data: data represented as numbers or discrete categories which can be directly mapped onto a numeric scale. **Quantitative analysis** consists of methods designed to summarise quantitative data.

Sample: A subset of the population.

Sampling frame: a list of all of the members of the population under investigation from which the sample is to be drawn.

Triangulation: the validation of a data item with the use of a second source, a second data collection mechanism, or a second researcher.

Type I error: incorrectly rejecting the true null hypothesis.

Type II error: incorrectly accepting the false null hypothesis.

Appendix B

Collected Interview Data

B.1 Structured interview template

Thanks for taking the time to be interviewed on this subject. The questions I'll ask will start off as relatively simple and should mainly be one or two sentence answers. Later questions move into more complicated areas and will mainly ask for opinions gained from actual experience, reading, and conferring with your colleagues.

1. To what extent is your knowledge of object oriented programming; that is how long have you used it, and how often do you use it?
2. Which language or languages are you most familiar with?
3. What size in executable lines of code are, on average, a single object's method (you may give a range if you wish)?
4. Compared to structured programs, what would you say the effect of changes are in object oriented programs. For example, are changes likely to cause 'ripples' to propagate through the system, or be localised to particular objects?
5. Do you use the inheritance facilities of your language(s)?

   No: why not?

   Yes: how often?
6. Do you think inheritance can cause difficulties when

   (a) trying to understand the code as a whole?

   (b) trying to understanding small segments of code (e.g., as a software maintenance engineer might do)?

No: what about when the hierarchy is particularly deep (define deep)?

No: why not?

Yes: see next question.

Yes: As a consequence do you think that software tools are a necessity for understanding and maintaining object-oriented programs.

No: why not?

Yes: Why? Any particular types you find useful, e.g., a cross reference browser, etc?

7. Inheritance can spread the functionality of particular methods over the system. What do you think is the impact of this from an understanding point of view? What do you think is the impact of this from a maintenance point of view?
8. What benefits, except for reuse, do you think inheritance provides?
9. What disadvantages do you think inheritance introduces (apart from the ones already discussed)?
10. The argument of inheritance versus multiple inheritance has existed for some time now. What are your opinions on this discussion? Which option do you think is best?
11. Without discussing inheritance, can object oriented code, in your opinion, cause problems to software maintenance engineers?

    Yes: Expand on these problems?

    No: What makes it easy to maintain then?

12. It is possible to maintain imperative code to death; that is the quality of the code deteriorates as it is continually maintained, until it becomes unmaintainable. Do you think that this can happen to object-oriented code, given changes are performed in an object-oriented fashion?

    No: why not?

    Yes: why?

13. How much use do you make of the class libraries? Which language?
14. Do you ever maintain these libraries by updating them?

    No: why not?

    Yes: What is the procedure for this?

15. Do you think as class libraries increase in size, and are used more often, the maintenance overhead spent on them will be greater than the maintenance overhead of the actual product? Opinions and explanations please.

16. Do you have any opinions on what is good or bad programming style when concerned with object oriented code quality? Has your company produced guidelines you should adhere to?

17. Given that a software maintenance engineer makes changes on a small scale, for example modifying a method, or adding a new method, is it possible to know whether the changes made have degraded the system code quality?

18. Would, in your opinion, a software maintenance engineer consult any design documentation to help him/her understand specific pieces of code causing difficulty?

19. Do you think that dynamic binding causes more problems than advantages?

    Yes: Expand on the specific disadvantages you feel are the worst.

    No: Expand on the specific advantages you feel are the best.

20. Polymorphism has been said to be a great strength of oo languages, but is also said to introduce difficulties in program analysis and understanding. Would you agree with this statement?

    Yes: Why?

    No: Why not?

21. Polymorphism, however, requires consistent use of method names within a system. Have you experienced any particular problems with consistent naming in this manner? What about the case of operator overloading?

    Yes: Why?

    No: Why not?

22. Are you aware of the C++ language?

    Yes: What do you feel are the disadvantages of this language when compared to other object-oriented languages? What advantages does C++ hold over other object-oriented languages.

    No: nothing further.

23. When a software maintenance engineer makes modifications is it possible that he might 'hack in a quick fix' instead of trying to maintain object-orientedness?

    Yes: Have you done this yourself? Have seen this done by any of your colleagues? Why is this quicker?

    No: Why not?

24. To conclude, do you believe that the object oriented paradigm, taking everything into consideration, is more beneficial than other paradigms?

25. Has there been anything that you have not been asked that you feel is important and should be addressed, or anything you would like to reiterate?

Thanks again for your time.

B.2 Paraphrased subject responses

This section details a summary of the remarks that individual subjects made to specific questions and issues. The data are presented in paraphrased form (although quotes indicates the subjects own words). No response means that the subject did not mention anything of significance. Tables are ordered by the questions in the interview template. Remarks made by a subject to a particular topic, but not given to a question are indicated and tabulated in the appropriate tables.

Subject	Remark
A	Small but up to 50
B	1-100
C	100+ (C-Flavours)
D	2-50
E	1-40
F	1-50; keep them small
G	1-30
H	1-20
I	5-10
J	6 lines
K	1-100
L	$<$ 12 lines
M	1 - screenful

Table B.1: Answers and remarks to Q. 4

Subject	Remark
A	Changes are localized, but hard to tell if system quality is degraded. Good design is the key
B	Depends on the design: changes localized if good design, but will affect the whole system if not
C	It depends on how well written and designed they are: less implications if this is done well, but difficult to maintain otherwise
D	Must be well designed: if they are then changes more localized
E	Localized, but design very important
F	Design very important: have classes correct (good abstractions) then more localized
G	Well designed code then changes are localized; bad code then they're not
H	Good design important: if good design then changes more localized
I	More modular code with object-orientation and less like to require changes
J	Localized, but can have a bigger global effect: affect all subclasses
K	More localized, design important
L	Localized, again design is important
M	Objects are structured pieces of code, more adaptable to change and easier to maintain

Table B.2: Answers and remarks to Q. 4 and various other questions

Subject	Remark
A	Tools are useful but not a necessity; class browser
B	Not a necessity, if you don't need tools to understand a non OO program then you don't need them for an OO program
C	Not a necessity, but useful
D	Not a necessity, but very helpful
E	Yes a necessity, you need them for visibility of whole maintenance process
F	Yes a necessity, but that's the case with everything
G	Not a necessity, but a definite benefit
H	Yes a necessity: it's hard to know the relationships between classes without them
I	Tools are very important; more are needed
J	I think they are a necessity, if dealing with something big then yes
K	A necessity: they help to visualize things you might not have realised
L	A necessity, yes I think so
M	A necessity, a good browser is essential

Table B.3: Answers and remarks from Q. 6

Subject	Remark
A	Makes it easier if abstractions are designed and implemented properly
B	It shouldn't make understanding difficult: inherited classes should implement same sort of function: understand the base class method to know what it's doing
C	Inheritance makes understanding easier, but it must be well designed and well written
D	Inheritance causes difficulties for tracing the flow of control particularly with multiple inheritance; "you don't know where you are going to end up."
E	Yes without fully appreciating repercussions in terms of dynamics
F	Yes, but only if it's wrong, if the inheritance is inappropriate
G	It means tracing through more code, but if done well then no problems
H	It can when methods are overloaded, e.g., where does this member variable get inherited? is hard to answer
I	No difficulty: can't see why it would
J	Can be difficult to trace which method is being called: tools help
K	Sometimes. Having to hunt through all the source code for implementation is problematic
L	Depth of inheritance an issue; easy to understand if designed well
M	related to depth, e.g., if you have to trace back through a great number of objects to see what method is implemented then yes

Table B.4: Answers and remarks to Q. 6

Subject	Remark
A	If using abstraction properly then a natural way of writing and understanding code
B	If designed properly then understanding ok
C	If designed and written well then ok, it's likely to aid understanding
D	No great impact on understanding
E	No more understanding difficulties than with structured programs
F	If the abstractions are clear then understanding should be ok
G	If done well then understanding should be ok
H	Inheritance simplifies things hence better understanding; "it simplifies your design."
I	No impact on understanding, it causes no difficulty
J	No problem, you don't have to deal with it at that level
K	No real problem
L	The hierarchy should be defined well so no understanding problem
M	I don't really know: it goes back to the design, the functionality spread should not affect me that much

Table B.5: Answers and remarks to Q. 7

Subject	Remark
A	If abstraction is enforced then it should be ok; if it isn't then there could be trouble; tracing of dependencies can be difficult but tools help
B	Can be difficult if inheritance is deep: difficult to trace dependencies
C	If well written then ok, life is easier
D	Good because fix it for one, fix it for all subclasses
E	Not qualified to answer
F	No experience of maintenance
G	Tracing up the hierarchy can be a problem
H	No more difficult than tracing C function calls, in fact its more structured so no specific problems; fix it for one fix it for all
I	No impact: programmers don't think like that
J	Depth of inheritance is an issue: the deeper the hierarchy the more tracing required therefore tools are important
K	Can cause problems because it may break subclasses who use the method that's been changed
L	No experience of maintenance
M	Inherent object-orientation has made it easy to the find way round a system, to find which bit was requiring maintenance

Table B.6: Answers and remarks to Q. 7

Subject	Remark
A	Able to design at highest level of abstraction
B	Provides modularity and prevents code redundancy
C	Aid to understanding and avoids code duplication
D	Simplifies control structure of methods
E	Provides focus for powerful abstraction
F	Reduces redundancy; allows abstraction
G	Helps produce nice designs at the conceptual level and aids understanding
H	No duplication of functionality, simplifies design
I	Reuse of specification
J	Information hiding and encapsulation
K	Better encapsulation, quick prototyping, high levels of abstraction, aid to understanding, simulates the real world
L	Quick prototyping, single inheritance good for design
M	"None, inheritance is a kind of implementation thing"

Table B.7: Answers and remarks to Q. 8

Subject	Remark
A	If the design is not good then understanding inheritance is a problem
B	Speed; the deeper hierarchy then the more complexity
C	If over used then it can result in code that's harder to follow
D	Tracing the flow of control
E	Misrepresenting abstractions is a problem
F	None
G	New concept to learn
H	None experienced
I	Multiple inheritance and inappropriate use of inheritance
J	Tracing dependencies; what method is this attached to? can be hard to answer
K	If not designed properly then BIG problems
L	Multiple inheritance
M	Losing your way in the hierarchy (related to depth)

Table B.8: Answers and remarks to Q. 9

Subject	Remark
A	Single inheritance far more useful
B	Multiple inheritance complicates things and software reuse is harder
C	Use multiple inheritance for complex applications; reuse and maintenance is easier with MI.
D	Multiple inheritance useful at the coding level
E	Multiple inheritance maps the reality of the system
F	Try to avoid multiple inheritance
G	Multiple inheritance a big problem - it's harder; SI is the way to do it
H	Multiple inheritance a trojan horse: it makes the design more complex
I	Multiple inheritance isn't worth the extra effort; it's more complex
J	Multiple inheritance comes very naturally
K	Multiple inheritance can be messy: it's an advantage but you must have a solid case for using it
L	Multiple inheritance allows for sloppy approach; you don't really need it more tightly coupled, and reuse is harder
M	"Multiple inheritance helped to underline the different functionality the object was inheriting"

Table B.9: Answers and remarks to Q. 10

Subject	Remark
A	Yes it can, but it's related to the learning curve: these do disappear
B	No more problems than structured code; difficult for those who don't know the code; splitting each class into a single file doesn't help
C	Problems if people doing maintenance don't have good object-oriented skills, but if these skills are possessed then it's easier
D	No problems, need knowledge of basic architecture but that's all; use of contractors easier for OO systems: they can get up to speed quicker
E	If only partial knowledge of the system then maintenance difficulties
F	No: it's the same as any other paradigm; abstraction aids maintenance
G	If personnel trained properly then they shouldn't have problems
H	Less problems that straight C: OO has more structure
I	If designed properly then objects shouldn't need maintenance, just specialization of existing classes
J	If poorly documented
K	One file per class idea can make it harder but tools help
L	Few systems mature enough for this to be decided
M	Can't envisage any great problems from it, my experience is than it is easier to maintain object-oriented code

Table B.10: Answers and remarks to Q. 11

Subject	Remark
A	"Getting into OO way of thinking takes time" "learning by osmosis"
I	"Learning curve is really fierce"
J	"You really have to think differently; I'm still thinking in structured C in some ways"
L	Almost 2 years to become a fully conversant object-oriented programmer

Table B.11: Remarks made about the learning curve

Subject	Remark
A	"Without a doubt" The rate of entropy leads to disorder and abstraction will eventually be lost
B	Quick fixes lead to unmaintainability.
C	Yes but process is slower than with structured programs.
D	Rate of entropy reduced: it's slower for object-oriented systems
E	There is a lesser tendency but must keep object-oriented mindset
F	Yes but depends on the design of the changes: must keep abstractions
G	Slower for object-oriented programs
H	If quick fixes unmaintainability ensues: corruption of the design
I	Much less tendency for object-oriented programs
J	Not as easily for object-oriented programs: changes made are object-oriented then the systems shouldn't become a mess
K	Less likely than with structured programs
L	Less so with object-oriented programs
M	Yes, I could envisage that situation occurring

Table B.12: Answers and remarks to Q. 12

Subject	Remark
A	A lot, all the time
B	Very limited use C++
C	None for C-flavours, **** for C++
D	Extensive in past, limited at present
E	None to date
F	Pretty limited
G	****
H	A lot for Objective-C, limited for C++: easier to write general purpose classes in Objective-C
I	A lot
J	Use bought libraries all the time
K	All the time; we build our own libraries
L	Extensive use of own libraries
M	Depends on what's available - beginning to in C++, all the time in Objective-C

Table B.13: Answers and remarks to Q. 13

Subject	Remark
A	Yes, take code from another programmer and modify it for new requirements; no for commercial libraries
B	No for commercial libraries; very limited maintenance and limited reuse otherwise
C	No for bought libraries, code is robust
D	Avoid it, perhaps specialization but not modifying the original code
E	No experience of maintaining class libraries
F	Definitely, change someone else's code to meet your requirements
H	A little maintenance but nothing major
I	No, class libraries tend to change very little
J	No
K	Yes, all the time through a revision control procedure
L	More so than a single programmer might do; trying to make code generic as possible in order to allow reuse
M	No

Table B.14: Answers and remarks to Q. 14

Subject	Remark
A	No, there is no distinction between libraries and application
B	Possible, but if well written then libraries don't need much maintenance
C	Less maintenance: likely to be more robust than new code if used before, but need good version and configuration control
D	No, being tested by a large user group and this irons out the bugs quickly: stability
E	Depends on scaleability of class libraries: need version and configuration control
F	Class libraries are static and hence reliable, so no
G	Class libraries are stable once they've been debugged, so no
H	In my experience class libraries once stable are changed very little
I	No, little maintenance of class libraries once stable
J	It's a possible scenario, but it's not happening at the moment
K	Problems get ironed out as you use it, so no
L	Shouldn't have to maintain libraries (Cox's maturity index a good idea): well used then well stressed and exercised
M	Depth of library an issue; incremental development with maturity index should prevent maintenance being required

Table B.15: Answers and remarks to Q. 15

Subject	Remark
A	Code must be put through code review, guidelines are subjective, e.g., don't use gotos, global variables, but bad design produces bad code
B	One file for one class is not good style; inconsistency
C	Long methods are a sign that something wants thinking about; excessive inheritance depth; "it's almost less the coding style than the design style that's important"
D	Short methods, protected instead of public inheritance where possible; not too deep an inheritance tree
E	I've not enough programming experience, it's vital that common programming guidelines are accepted though
F	Good design usually means good code: we have code reviews earlier now
G	I don't like overloading, it complicates things too much
H	Should have code reviews to sort things out, seems to be few points of reference for OO code quality
I	Identification of correct objects; inappropriate use of inheritance: all at the design level
J	Small methods; mixing object-oriented code with procedural code is sloppy
K	"Mechanism not policy" (based on good design)
L	Question not asked
M	Good design then good code, bad design then bad code

Table B.16: Answers and remarks to Q. 16

Subject	Remark
A	Difficult to say, changes should be localized: only when encapsulation is broken is there a worry, code reviews help
B	Difficult to know, code reviews help
C	Its no more impossible than under the procedural type of method; breaking encapsulation is a bad idea, code reviews help
D	No, but this is true of any system
E	It depends on the utility of the method changed
F	Use metrics and code reviews to check so yes it is possible
G	No, but a code review should spot it
H	It's subjective, but code reviews spot these things
I	Question not asked
J	Yes, the changes should be tested adequately and encapsulated
K	No, difficult to tell; code reviews help
L	Encapsulation so it should not have degraded
M	Very difficult to tell

Table B.17: Answers and remarks to Q. 17

Subject	Remark
A	Trouble with availability and quality of design documentation Subjects B, C, D, F, G, H, J, K, & M made similar statements
E	Documentation is usually available
G	"Design documentation is important otherwise you are back to where you were before"
L	Documentation for OO programs more important aid to understanding than documentation for non OO systems
M	"The documentation is essential"

Table B.18: Answers and remarks to Q. 18

Subject	Remark
A	No, without dynamic binding no object-oriented programming
B	No, really useful; Objective-C far better than C++ for it
C	No, it allows very elegant code, but there can be problems of finding methods bound at run-time
D	No, it's a big help: simplifies the code; it's more intuitive
E	No, if managed well it maps reality
F	No, good/clear abstractions are aided by it
G	If you understand it, then it causes few problems
H	No, it makes your design simpler
I	Question not asked
J	No, it allows generic code, no switches etc
K	No, it provides lots of advantages
L	No, only problem is calling a non existent method
M	Yes if you are careless, no if you are not

Table B.19: Answers and remarks to Q. 19

Subject	Remark
A	No, using a message should have the same effect on any object that responds to it; abstraction
B	Yes, it's a great benefit, but it can make things difficult to understand. If used properly and sparingly it's advantageous
C	Yes, it produces generic code, but easy to lose track of what's going on
D	No, it simplifies things rather than making them more difficult: you are looking at the behaviour of each object but not the details
E	Maintain semantic consistency and no problem, otherwise problems
F	No, methods with same name should achieve the same aim
G	Yes but related to learning the concept
H	It simplifies things; I think it's wonderful
I	It's the most important property in OO systems, but must maintain high level semantics
J	It can cause confusion: you just have to be careful
K	No problem, it improves understanding
L	It's an aid to human understanding, it's more intuitive
M	No problem with the size of systems I've looked at

Table B.20: Answers and remarks to Q. 20

Subject	Remark
A	No problems, perhaps with operator overloading, but tools help this
B	No problems, but I avoid operator overloading as it is not predictable
C	No problems, no operator overloading experience
D	Yes, naming is big issue in object-oriented programming, no great problems with operator overloading, but no great experience either
E	No real experience of any such problems
F	Not with polymorphism, but experience of operator overloading not pleasant
G	No, but little use; operator overloading used sensibly then useful
H	Yes, it's hard to know the best time to use the same names
I	This is one of the major hidden problems in object-oriented systems, programmers must understand about consistent naming. Operator overloading for thoroughly defined operators only
J	None experienced; no use of operator overloading
K	Yes, easily make a mistake renaming a member function, and this can lead to subtle errors that are hard to find because everything looks ok
L	Not personally
M	No

Table B.21: Answers and remarks to Q. 21

Subject	Remark
A	It's been hacked together without much thought
B	Syntax of it, can be very obscure
C	Same pitfalls as C, easy to write C in C++. "Regress back into doing something in a C fashion when they should be doing it in OO fashion"
D	Complicated, tries to do everything, inconsistent
E	Unit of modularity, the class, has not been scaled up
F	Same problems as C, strong typing, write C instead of C++
G	All of the C problems
H	Huge monster of a language, strongly typed, obscure
I	Time bomb on wheels, allows bad programmers to write bad object-oriented code
J	Almost programming in C; mixing and matching paradigms is sloppy
K	Undefined areas (e.g., protected inheritance), doesn't look object-oriented, doesn't encourage object-oriented way of thinking
L	Doesn't encourage object-oriented programming: you may write C instead of C++; it's strongly typed
M	Strong typing; sold as a better C: allows you to feel as if you are an object-oriented programmer, but you are not, allows you to write C; not forced to write object-oriented code

Table B.22: Answers and remarks to Q. 22

Subject	Remark
A	Its efficiency and popularity
B	Its fast
C	Its performance
D	Familiarity with C and speed
E	Difficult to say if it has any
F	Control over everything
G	Its performance
H	Its performance
I	Allows use existing C libraries and OO is taking off because of C++
J	Its efficiency; existing C libraries
K	More efficient
L	Efficiency and it's street credibility
M	Efficient; backed by AT&T

Table B.23: Answers and remarks to Q. 22

Subject	Remark
A	"Yes, without a doubt. It's common practice" "Time pressure does exist"
B	It depends on the maintainer. It's "easier in C++"
C	"Entirely possible" "Only thing you can do in the time"
D	"Absolutely, someone pushed in a hurry will do it the easiest way"
E	"It's too easy in C++"
F	Yes because of time pressures
G	"Yes, but that's bad maintenance rather than design"
H	"There's a good chance it might happen" I've only done it once
I	"Yes particularly in C++"
J	In C++ a lot easier to do quick fix; depends on nature of the project as well as individual
K	"Yes absolutely"
L	"One of the dangers in these hybrid languages"
M	"It's all too easy, it's really easy, and the temptation is always there

Table B.24: Answers and remarks to Q. 23

Subject	Remark
A	time constraints have caused quick fixes to OO code in the past Subjects C, D, F, and K made similar statements
D	Must try to remain competitive and that can cause quick fixes
K	Documentation is the first thing that suffers because of time constraints

Table B.25: Remarks about time constraints when performing quick fixes

Appendix C

Collected Questionnaire Data

C.1 Questionnaire on object-oriented systems

Section 1 — Your Background with Object-Oriented Technology

Base your answers to the questions in Section 1 on experience with object-oriented technology. CIRCLE the category which most accurately describes your answer. Please feel free, however, to articulate information which you regard as relevant at the end of each question.

1. (a) What is your current position?
   - Student
   - Academic
   - Software Engineer
   - Project Manager
   - Other - please specify:

   (b) In what capacity do you use object-oriented technology?
   - Teaching
   - Programming
   - Analysis and design
   - Other - please specify:

2. (a) How long have you used object-oriented technology?
   - < 1 year
   - 1 - 2 years
   - 3 - 4 years
   - > 4 years

   (b) How often do you use object-oriented technology (e.g., every day, once a week, etc)?

3. Which object-oriented language(s) are you most familiar with? (you may circle more than one category)

   C++
   Objective-C
   Eiffel
   Smalltalk
   CLOS
   Other - please specify:

4. A method is typically how many executable lines of code?

   1 - 4 lines
   5 - 10 lines
   11 - 20 lines
   > 20 lines

   Your range of smallest to largest is:

5. How deep would your inheritance hierarchy be before you became uncomfortable with it?

   1 level
   2 - 3 levels
   4 - 6 levels
   > 6 levels
   No problem with depth

6. What causes you the most difficulty when trying to understand an object-oriented program?

   Please specify:

7. Has inheritance caused difficulty when trying to understand an object-oriented program? (please circle the appropriate number)

   Never Always
   1 2 3 4 5

8. Overloading requires consistent use of method names within a system. Have you experienced any problems with consistent naming in this manner (i.e., methods did not maintain semantic consistency)? (please circle the appropriate number)

   Never Always
   1 2 3 4 5

9. Have you made use of class libraries that are local to your company/academic institution (i.e., designed and implemented by your company/institution)?
   (please circle the appropriate number)

   Never Always
   1 2 3 4 5

Section 2 — Opinionated Object-Oriented Questions

Base your answers to Section 2 on your opinions. Opinions can be gauged on experience, reading or conferring with colleagues. Again, feel free to articulate any relevant information at the end of each question.

10. Do you believe that the object-oriented paradigm is more beneficial than other paradigms in terms of ...

    Ease of analysis and design: Yes
    No
    Don't know

    Programmer Productivity: Yes
    No
    Don't know

    Software Reuse: Yes
    No
    Don't know

    Ease of Maintenance: Yes
    No
    Don't know?

    Any other reasons? - please specify:

11. Is multiple inheritance useful? (please circle the appropriate number)

    Never Always
    1 2 3 4 5

12. Do you think that continual maintenance of structured programs, i.e., non object-oriented programs, leads to unmaintainability? (please circle the appropriate number)

    Never Always
    1 2 3 4 5

13. Do you think that continual maintenance of object-oriented programs leads to unmaintainability? (please circle the appropriate number)

    Never Always
    1 2 3 4 5

14. Do you think that object-oriented code is more maintainable than structured code? (please circle the appropriate number)

    Never Always
    1 2 3 4 5

Please answer questions 15 to 19 only if you have knowledge of C++ or strong opinions on the subject in question.

15. C++ appears to have become the de facto standard object-oriented language for industry.
    Do you regard this as
    Bad
    Good
    Indifferent
    Don't know, or
    Disagree with statement, please say why:

16. C++ allows a mixture of object-oriented programming and structured programming.
    Do you see this mixture of paradigms as an
    Advantage, please say why:

    Disadvantage, please say why:

    Don't know?

17. When maintaining a C++ program would you make use of the FRIEND function rather than redesign your inheritance hierarchy and thus maintain object-orientedness?
    (please circle the appropriate number)
    Never Always
    1 2 3 4 5

18. (a) How often do you make use of operator overloading?
    (please circle the appropriate number)
    Never Always
    1 2 3 4 5

    (b) Do you overload operators as
    Member functions, please say why:

    Non-member functions, please say why:

    Both of the above, please say why:

    Not Applicable?

19. How often do you make use of templates?
    (please circle the appropriate number)
    Never Always
    1 2 3 4 5

Comments and points you would care to elaborate on:

C.2 Raw questionnaire data

This section provides the raw data for the 275 completed questionnaires. The data is is ordered by question and is coded as follows: a 0 indicates no response was given for that question, a 1 indicates the first category was chosen for the answer to the question, a 2 indicates the second category was chosen, and so forth. In the case where the number listed is outwith the number of categories available, a combination of categories where chosen by the respondent. Finally, the last number of each row indicates which distribution media the questionnaire was received from: 0 ≡ electronic newsgroups and 1 ≡ postal.

```
1  8  3  1  6   2  2  5  2  4  2  3  1  1  1  0  0  3  3  1  0  0  0  0  0  0
1  8  4  3  10  0  1  5  3  4  5  3  1  3  0  4  0  0  4  1  3  3  2  3  4  0
2  2  2  3  6   3  1  5  2  3  1  3  1  1  1  2  2  2  4  0  0  0  0  0  0  0
3  2  3  1  7   3  2  3  3  2  2  3  3  1  1  2  4  3  4  1  2  2  1  4  1  0
3  8  3  1  8   2  1  5  1  2  3  3  1  1  1  0  4  3  4  2  2  5  0  0  0  0
3  2  3  1  1   1  3  5  1  2  5  3  1  1  2  4  2  2  4  6  1  1  1  4  1  0
3  2  3  1  8   3  0  5  4  2  4  3  1  1  1  1  4  3  5  1  1  1  2  0  1  0
4  2  2  1  1   3  3  3  2  1  4  3  1  1  1  3  4  2  4  2  4  3  3  3  4  0
2  5  2  3  1   3  0  5  3  1  1  3  1  1  3  5  3  3  3  1  2  0  5  3  5  1
2  7  3  1  13  3  2  5  2  3  2  3  3  1  1  3  4  2  4  1  2  4  4  1  3  1
3  8  1  1  4   2  1  3  2  2  1  3  1  1  1  4  2  0  0  0  0  0  0  0  0  1
3  8  1  1  4   0  0  5  2  0  1  3  3  3  3  3  0  0  0  1  2  0  0  0  0  1
3  2  1  1  4   2  1  3  2  3  4  3  1  1  3  3  4  3  3  0  0  0  0  0  0  1
3  2  1  1  1   0  0  4  2  1  1  3  2  1  1  4  3  2  4  3  2  4  1  4  4  1
3  2  2  1  6   0  0  2  3  1  1  3  1  1  1  1  5  0  4  3  1  1  0  0  0  1
3  2  3  1  4   3  1  5  2  2  5  3  1  1  3  3  0  2  5  0  0  0  0  0  0  1
3  2  3  1  1   3  2  5  2  1  4  3  1  1  1  4  2  2  4  2  4  2  5  1  1  1
4  2  4  1  4   3  1  5  2  1  1  3  1  1  3  2  4  3  4  0  0  0  0  0  0  1
5  4  0  0  12  0  0  3  3  2  0  3  3  1  1  3  3  2  2  1  2  2  0  0  0  1
5  7  2  1  4   1  0  5  2  3  3  3  1  1  1  2  4  2  4  0  0  0  0  0  0  1
5  7  4  1  1   1  3  4  1  2  4  3  3  1  1  3  4  3  5  2  1  1  4  3  4  1
1  2  2  0  1   0  1  3  2  1  2  2  1  1  1  0  5  3  5  2  3  0  3  1  2  0
1  2  2  1  16  3  1  5  2  3  2  2  1  1  1  2  3  2  4  1  2  3  5  1  4  0
1  2  3  2  16  2  2  5  2  4  4  2  1  2  1  4  2  3  4  1  1  4  4  3  1  0
2  5  4  1  7   4  3  5  2  3  4  2  1  1  3  2  4  2  4  1  2  1  1  4  4  0
2  1  4  3  9   2  0  3  2  1  2  2  1  1  1  1  0  2  5  1  2  1  3  1  1  0
3  2  2  1  5   3  1  5  4  2  3  2  1  1  1  4  5  5  4  5  1  0  0  0  0  0
3  8  2  1  7   2  2  2  3  4  5  2  1  1  1  2  5  2  4  1  2  4  4  0  1  0
3  8  2  1  1   4  5  3  3  0  2  2  1  2  0  0  4  0  3  1  2  0  1  0  1  0
3  8  3  1  1   2  2  3  2  1  4  2  2  2  3  2  2  2  2  1  1  2  2  3  3  0
3  8  4  2  1   0  1  5  2  2  2  2  2  2  2  3  3  3  3  1  1  2  3  1  1  0
5  7  4  1  9   3  2  5  3  4  5  2  1  1  2  3  2  2  3  1  2  3  4  3  1  0
1  2  3  1  14  3  1  4  1  3  4  2  1  1  1  1  4  2  5  1  2  0  0  0  0  1
2  1  3  1  13  3  1  5  3  4  0  2  1  3  3  3  5  3  4  1  1  1  1  4  1  1
2  1  4  3  14  2  0  3  2  1  2  2  1  1  1  1  0  2  5  1  2  1  3  1  1  1
3  4  1  2  1   2  1  2  3  1  1  2  2  1  1  2  3  3  3  2  1  2  2  2  1  1
3  2  1  2  14  2  0  3  3  4  1  2  2  2  2  0  2  2  3  0  0  0  0  0  0  1
3  2  1  3  1   3  3  3  3  2  3  2  1  1  2  3  3  3  3  2  1  1  2  1  1  1
3  2  2  2  6   4  5  5  3  4  1  2  2  1  1  4  5  3  4  1  1  3  1  4  4  1
3  8  3  1  14  3  0  3  4  4  5  2  1  2  1  2  4  3  3  1  2  2  1  4  5  1
3  8  3  1  14  3  1  4  2  2  2  2  1  3  2  4  2  4  1  1  2  3  2  1  4  1
3  2  4  1  16  2  2  3  2  2  1  2  1  1  1  3  3  3  3  1  2  2  2  4  3  1
3  8  4  1  4   1  1  5  1  3  1  2  1  1  1  1  3  2  4  5  2  0  0  0  0  1
4  2  3  1  14  3  3  3  2  1  5  2  1  1  1  1  3  3  3  3  3  1  4  1  2  1
4  3  3  2  14  3  1  5  2  3  4  2  2  1  1  2  4  3  4  1  2  2  2  0  3  1
4  3  4  1  14  2  0  3  5  4  5  2  1  1  2  1  5  5  1  2  1  5  5  3  3  1
5  8  4  2  14  2  1  5  2  1  3  2  2  1  2  2  2  2  4  1  4  4  2  1  2  1
```

5	1	4	1	14	3	4	3	2	1	2	2	1	1	1	3	5	2	4	2	1	2	3	3	3	1
1	8	1	1	1	3	0	5	1	1	1	1	3	3	3	2	4	3	4	3	3	4	5	1	4	0
1	2	1	1	11	3	1	5	3	1	3	1	2	1	1	5	2	2	4	2	1	2	5	1	1	0
1	8	1	4	1	3	0	5	1	3	1	1	1	1	1	4	3	2	4	2	1	2	3	1	4	0
1	3	2	1	16	1	1	4	2	1	1	1	1	1	1	3	4	2	4	1	2	1	4	2	4	0
1	2	2	1	2	1	1	5	4	1	1	1	1	1	1	2	0	1	5	1	4	2	1	4	1	0
1	8	2	2	16	2	1	2	3	4	3	1	1	3	1	3	2	3	4	0	0	0	0	0	0	0
1	2	2	1	1	3	1	3	1	2	3	1	1	1	3	2	0	0	0	2	3	4	2	1	1	0
1	8	2	1	1	4	1	5	5	4	1	1	1	1	3	0	3	0	0	1	0	4	4	3	1	0
1	2	2	1	5	2	0	3	2	3	1	1	1	1	1	4	3	2	4	0	0	0	0	0	0	0
1	2	2	1	4	2	1	2	2	2	2	1	1	1	1	2	4	2	4	0	0	0	0	0	0	0
1	8	2	0	11	2	1	3	2	3	1	1	1	1	1	5	0	0	4	1	4	3	1	4	1	0
1	2	3	1	7	2	2	3	2	2	2	1	2	1	1	2	4	3	4	1	1	1	5	2	4	0
1	2	3	2	2	3	1	3	4	2	2	1	1	1	1	1	5	3	5	1	1	0	0	0	0	0
1	2	3	1	7	1	1	5	2	2	1	1	1	3	1	2	4	2	5	1	1	3	4	3	5	0
1	8	3	1	9	2	0	5	2	3	2	1	1	1	1	0	4	4	0	1	2	3	1	4	1	0
1	8	3	1	2	3	4	4	2	1	4	1	1	1	1	1	4	2	5	0	0	0	0	0	0	0
1	8	3	1	16	2	2	5	2	2	2	1	1	1	1	2	0	0	4	1	3	3	2	1	5	0
1	7	3	1	4	3	2	2	0	1	2	1	1	1	2	2	4	4	0	1	1	4	4	3	2	0
1	8	3	1	4	2	1	5	2	2	1	1	1	1	0	2	0	0	0	0	2	0	0	0	0	0
1	8	3	1	16	3	1	3	3	1	1	1	1	3	4	4	2	4	1	4	3	1	0	4	0	
1	2	3	3	4	2	0	4	2	2	1	1	1	1	2	3	5	4	3	1	2	2	2	4	1	0
1	5	3	1	4	2	1	5	2	0	2	1	1	3	3	4	0	0	4	4	2	0	0	0	0	0
1	3	3	1	16	0	0	3	4	3	1	1	3	3	3	3	4	2	0	1	2	0	0	0	0	0
1	3	3	1	16	2	0	5	2	1	2	1	3	1	1	3	0	0	4	1	2	1	4	0	4	0
1	3	3	1	16	0	0	3	4	3	1	1	3	3	3	3	4	2	0	1	2	0	0	0	0	0
1	2	4	1	10	3	0	5	2	2	5	1	3	1	1	5	5	2	5	1	3	2	2	0	2	0
1	5	4	1	3	2	1	5	1	0	5	1	1	1	1	5	4	2	4	1	2	0	0	0	0	0
2	2	1	2	11	2	0	3	4	2	1	1	1	1	1	0	2	2	4	1	3	1	2	1	3	0
2	5	1	3	2	2	1	3	2	1	2	1	1	1	1	0	4	2	4	1	1	0	2	4	2	0
2	5	2	1	9	2	1	3	3	2	3	1	1	1	4	3	0	4	1	2	1	2	1	3	0	
2	4	2	1	9	2	0	3	2	0	1	1	3	2	1	3	4	3	4	1	2	1	1	0	2	0
2	1	3	1	6	2	1	5	0	1	3	1	1	3	1	2	3	2	4	1	2	0	0	0	0	0
2	8	3	1	2	3	2	3	2	1	1	1	1	1	1	1	3	2	4	1	2	0	1	4	1	0
2	4	3	1	10	3	2	2	4	5	1	1	1	1	1	3	3	3	5	1	2	4	1	4	2	0
2	8	3	2	2	3	5	3	1	3	4	1	1	1	1	1	4	2	5	1	4	0	0	0	0	0
2	4	3	1	10	3	2	2	4	5	1	1	1	1	1	3	3	3	5	1	2	4	1	4	2	0
2	5	4	2	4	2	0	5	3	2	4	1	1	2	2	1	4	0	3	0	0	0	0	0	0	0
2	5	4	1	7	2	1	4	2	4	4	1	1	1	1	5	4	4	5	1	2	1	1	4	5	0
2	6	4	1	16	0	0	3	3	2	2	1	1	1	3	2	5	5	4	1	1	0	0	0	0	0
2	7	4	1	16	1	1	3	4	2	4	1	1	1	5	5	3	4	1	2	1	1	4	2	0	
2	7	4	1	6	4	0	5	3	0	2	1	1	1	1	3	4	2	4	1	1	0	0	0	0	0
2	7	4	1	10	3	2	5	1	3	5	1	1	1	1	5	4	2	4	1	1	0	0	0	0	0
2	7	4	2	7	1	1	5	2	4	3	1	1	1	2	2	4	4	2	5	2	0	0	0	0	0
2	7	4	1	0	0	0	2	3	3	1	1	2	1	2	4	3	3	3	1	2	3	3	3	3	0
2	7	4	3	3	3	0	4	3	2	4	1	1	1	1	5	5	2	4	1	2	3	1	4	3	0
2	5	4	1	7	2	0	2	1	2	4	1	1	1	1	1	4	2	5	1	2	1	3	4	0	0
2	1	4	1	9	3	1	3	4	3	1	1	1	1	3	3	0	0	4	1	2	4	4	3	2	0
3	2	1	1	7	4	2	5	3	4	4	1	1	1	1	3	2	3	4	1	2	2	2	1	1	0
3	2	1	1	1	3	4	5	1	3	4	1	1	3	1	0	5	2	5	2	4	1	4	3	1	0
3	2	1	1	1	4	3	5	1	2	5	1	1	1	1	1	1	0	3	2	1	1	5	3	1	0
3	2	2	1	7	3	2	3	2	2	3	1	1	1	1	2	4	2	4	1	0	2	2	0	1	0
3	2	2	1	2	3	1	3	2	0	2	1	1	1	1	0	3	2	4	1	1	0	0	0	0	0
3	2	2	1	2	3	2	5	1	3	3	1	1	1	1	4	5	2	4	0	0	0	0	0	0	0
3	2	2	1	2	2	2	3	3	3	2	1	1	1	3	2	4	4	2	1	2	0	0	0	0	0
3	2	2	1	6	3	4	5	2	4	5	1	1	1	1	0	2	2	4	0	0	0	0	0	0	0
3	8	2	1	7	3	0	4	2	2	4	1	1	1	1	4	4	3	4	3	2	2	5	3	5	0
3	2	2	1	8	3	0	5	1	3	2	1	2	1	1	5	3	1	5	3	1	1	1	4	2	0
3	2	2	2	2	2	0	4	3	3	4	1	1	1	1	2	4	3	5	1	2	1	1	4	2	0
3	8	2	1	3	3	2	3	2	2	3	1	1	1	1	3	4	3	4	1	2	0	0	0	0	0
3	7	2	1	1	2	1	5	2	2	1	1	2	1	1	3	4	2	4	3	4	2	2	2	3	0
3	2	2	1	8	0	2	5	3	2	4	1	1	1	1	5	2	3	5	1	4	2	2	1	3	0
3	8	2	1	8	3	5	3	4	3	4	1	1	2	1	2	2	3	4	1	1	3	2	1	1	0
3	8	2	1	1	4	5	3	1	1	1	1	1	1	1	1	3	3	3	1	4	3	3	2	1	0
3	8	2	1	1	3	1	3	4	1	5	1	3	1	1	2	5	3	3	1	4	1	2	1	1	0

3	8	2	1	11	2	2	2	3	4	3	1	1	1	1	2	3	3	3	1	1	4	2	1	1	0
3	8	2	3	7	3	1	2	1	2	1	1	1	1	1	3	3	3	4	3	4	3	3	4	1	0
3	8	2	2	6	3	2	5	4	2	5	1	1	1	1	0	3	3	4	1	2	4	4	0	0	0
3	2	2	2	1	0	1	4	2	0	5	1	1	1	3	5	3	0	0	4	3	4	0	1	1	0
3	8	3	1	7	2	3	5	2	2	5	1	1	1	1	4	4	2	5	6	1	3	3	4	1	0
3	2	3	1	1	3	1	3	4	2	2	1	3	1	1	3	3	3	4	3	2	2	4	1	2	0
3	8	3	1	1	3	1	5	2	2	4	1	1	1	1	3	4	2	4	3	2	1	2	1	3	0
3	8	3	1	16	3	0	3	3	4	5	1	1	1	1	2	4	3	4	1	2	2	1	4	1	0
3	2	3	1	1	4	4	2	3	2	2	1	1	1	1	0	3	2	5	2	1	1	3	1	1	0
3	8	3	1	9	1	1	3	1	2	3	1	1	1	1	3	2	3	5	0	0	0	0	0	0	0
3	3	3	1	1	4	2	4	2	1	1	1	1	1	1	4	4	3	4	2	1	2	2	0	1	0
3	7	3	1	3	3	4	5	2	0	4	1	1	1	1	5	3	1	4	2	0	0	0	0	0	0
3	8	3	1	1	3	1	3	3	2	3	1	1	1	1	2	4	2	4	2	4	1	2	1	1	0
3	7	3	1	1	3	1	5	2	2	4	1	1	1	1	3	4	2	4	2	1	1	4	1	1	0
3	8	3	1	1	2	4	4	1	2	4	1	1	1	1	2	4	3	4	6	4	2	4	3	1	0
3	7	3	1	4	2	1	3	2	2	4	1	1	1	2	4	3	3	3	1	2	3	5	4	0	0
3	8	3	1	9	2	1	5	1	1	3	1	1	1	1	3	4	3	4	3	2	2	4	3	3	0
3	2	3	1	10	2	1	5	2	2	3	1	1	1	1	5	4	2	5	1	2	2	2	0	2	0
3	8	3	1	4	3	2	3	2	3	3	1	1	1	1	2	4	2	5	0	0	0	0	0	0	0
3	8	3	1	1	1	2	5	2	4	1	1	1	1	1	2	2	2	4	2	1	3	2	3	4	0
3	8	3	0	16	3	1	3	4	3	4	1	1	1	1	5	5	5	5	2	2	1	2	2	2	0
3	2	3	1	1	3	2	5	2	2	4	1	1	2	3	3	0	0	4	2	1	2	3	3	2	0
3	8	3	1	16	3	1	5	2	3	4	1	1	1	1	2	3	3	4	1	1	2	2	1	1	0
3	8	3	1	3	2	5	5	2	1	5	1	2	2	1	5	4	4	3	1	2	2	0	0	0	0
3	8	3	1	7	2	1	5	2	1	4	1	1	1	3	0	4	2	5	5	3	2	5	3	1	0
3	3	3	1	9	2	0	5	1	1	3	1	1	1	1	1	5	3	4	3	2	3	3	4	4	0
3	8	3	1	9	2	1	5	1	2	2	1	1	2	2	2	4	3	4	6	1	2	3	4	0	0
3	8	3	1	2	4	1	3	2	0	3	1	1	1	1	4	4	2	4	0	0	0	0	0	0	0
3	2	3	1	7	3	5	3	4	4	2	1	1	2	3	2	4	2	5	2	3	2	4	1	2	0
3	8	4	1	1	4	2	5	2	2	2	1	1	1	1	3	5	2	5	2	1	1	4	3	3	0
3	8	4	1	1	4	5	3	1	1	5	1	1	2	1	3	3	3	3	1	1	3	5	3	4	0
3	2	4	1	5	2	5	5	2	2	5	1	1	1	1	5	4	2	4	1	1	2	0	0	0	0
3	7	4	1	4	3	2	3	3	4	5	1	1	1	1	0	2	5	4	1	4	0	0	0	0	0
3	5	4	1	10	3	1	5	2	2	4	1	1	1	1	5	4	3	4	5	1	3	4	1	1	0
3	2	4	1	16	4	1	5	2	2	3	1	2	1	1	4	5	2	4	3	2	5	4	3	2	0
3	7	4	1	9	4	2	5	1	1	2	1	1	1	1	0	4	2	5	3	1	1	5	3	2	0
3	8	4	1	16	4	5	4	2	2	4	1	3	1	1	4	3	3	4	1	2	4	1	4	4	0
3	8	4	1	1	2	1	4	2	1	3	1	1	3	1	3	4	3	2	2	1	2	3	1	2	0
3	7	4	2	10	2	1	4	1	2	4	1	1	1	1	3	0	0	3	1	1	2	2	0	1	0
3	2	4	1	16	3	2	3	2	3	4	1	1	1	1	2	3	2	4	1	1	1	1	4	1	0
3	8	4	1	9	2	1	4	3	2	4	1	1	1	1	2	4	3	4	1	3	4	2	1	1	0
3	8	4	1	16	1	5	4	3	2	5	1	2	1	1	3	4	3	4	1	4	1	3	3	3	0
3	8	4	1	9	2	1	5	2	1	5	1	1	1	1	4	4	3	4	1	2	2	3	3	1	0
3	3	4	1	1	1	4	5	2	3	5	1	1	1	1	2	4	2	4	3	1	2	4	1	1	0
3	8	4	1	1	3	4	4	2	1	4	1	3	3	1	3	5	5	4	3	4	3	2	2	1	0
3	8	4	1	16	2	1	3	1	2	4	1	1	1	1	4	5	2	5	1	2	3	4	3	3	0
3	8	4	1	6	4	0	3	2	2	2	1	1	1	3	4	2	2	2	1	1	0	3	0	3	0
3	8	4	1	10	3	2	5	2	2	4	1	1	3	1	4	3	2	4	1	1	2	2	3	4	0
3	7	4	1	16	1	1	3	3	2	3	1	2	2	2	1	3	3	3	1	1	2	2	3	2	0
3	8	4	1	10	3	5	3	2	2	4	1	1	1	1	4	5	5	4	6	1	3	4	3	4	0
3	8	4	1	7	0	1	5	4	3	2	1	1	1	1	2	5	3	4	3	2	2	3	1	4	0
3	8	4	1	10	2	1	5	2	2	1	1	1	1	1	4	4	3	4	1	2	2	2	0	5	0
3	7	4	1	4	2	1	5	1	2	2	1	1	1	1	2	5	5	4	5	2	0	0	0	0	0
3	8	4	1	16	3	0	5	1	1	2	1	1	1	1	2	4	3	5	1	1	2	4	0	2	0
3	8	4	1	4	3	1	5	2	2	5	1	1	1	1	2	4	2	5	1	2	0	0	0	0	0
4	2	3	1	5	2	0	5	2	1	4	1	1	3	1	5	3	3	4	0	0	0	0	0	0	0
4	8	3	1	2	3	2	5	2	2	2	1	1	1	1	1	5	3	5	1	1	0	0	0	0	0
4	8	3	1	2	2	1	4	0	2	4	1	1	1	1	1	4	2	4	1	1	2	2	0	3	0
4	7	3	1	16	3	0	2	1	1	5	1	1	1	1	4	4	4	3	1	2	1	3	4	2	0
4	8	4	1	7	0	4	5	1	3	4	1	1	1	1	3	3	1	1	1	1	2	2	3	2	0
4	8	4	1	9	3	2	5	1	1		1	1	1	2	3	4	3	3	1	4	2	5	1	3	0
4	8	4	1	16	4	0	5	2	3	5	1	1	1	1	5	3	3	5	1	2	2	1	4	1	0
4	8	4	2	1	2	1	5	1	2	1	1	1	1	1	5	5	1	5	2	2	1	3	3	4	0
4	8	4	1	1	3	0	3	3	3	4	1	1	1	1	3	2	2	4	2	1	2	3	0	1	0
5	2	2	2	1	3	2	2	2	1	5	1	1	1	3	0	4	2	4	2	1	3	4	1	1	0

5	2	3	1	8	0	0	5	1	2	4	1	1	1	3	4	4	3	3	1	2	2	2	3	4	0
5	4	3	1	8	0	0	5	1	2	4	1	1	1	3	4	4	3	3	1	2	2	3	3	4	0
5	3	3	1	8	3	0	3	2	2	5	1	3	1	1	3	3	0	4	3	1	2	3	1	3	0
5	7	4	1	9	1	0	5	1	2	5	1	1	1	1	3	2	2	4	1	4	0	1	4	1	0
5	2	4	1	7	2	1	5	2	2	4	1	1	1	1	2	3	2	4	1	1	0	0	0	0	0
5	4	4	1	4	2	1	5	2	2	5	1	1	1	1	2	5	3	4	1	2	0	0	0	0	0
5	1	4	1	16	2	0	5	2	2	3	1	1	1	1	3	3	2	4	1	2	2	3	1	3	0
5	8	4	1	1	0	1	3	2	2	4	1	1	1	1	3	4	2	4	2	1	2	3	3	1	0
5	7	4	1	9	0	1	5	4	3	3	1	1	1	1	2	2	2	3	3	1	2	2	3	4	0
5	7	4	1	4	2	0	5	1	2	3	1	0	3	1	0	0	0	0	1	2	0	0	0	0	0
5	7	4	1	4	1	1	5	2	3	3	1	1	1	2	4	3	4	1	4	0	0	0	0	0	0
5	2	4	1	16	3	2	5	2	1	1	1	1	1	1	5	4	4	4	1	1	0	0	0	0	0
5	7	4	1	11	2	4	3	3	4	4	1	1	2	1	1	0	0	4	1	2	2	5	1	4	0
1	3	1	1	1	0	0	2	2	1	4	1	1	1	1	4	4	2	4	1	2	0	0	0	0	1
1	8	2	3	1	4	4	2	3	1	1	1	3	1	1	4	5	2	5	2	1	4	2	0	1	1
2	1	3	3	1	2	1	2	1	1	3	1	3	1	1	3	3	0	4	3	1	1	3	1	1	1
2	4	3	1	4	2	1	4	2	3	1	1	1	1	1	2	4	2	5	0	0	0	0	0	0	1
2	5	3	4	6	2	1	3	0	0	1	1	3	1	1	0	4	2	4	1	2	0	0	0	0	1
2	8	4	1	4	3	4	5	1	3	1	1	1	1	1	4	3	3	4	1	2	2	4	0	1	1
2	2	4	2	4	3	1	3	2	4	4	1	1	1	1	3	4	3	4	1	1	0	0	0	0	1
2	1	4	1	4	1	1	5	4	4	2	1	2	1	3	3	3	2	4	1	2	0	4	3	1	1
2	1	4	1	16	2	0	5	3	3	1	1	1	1	3	4	4	4	0	1	3	1	5	1	3	1
3	8	1	1	3	0	0	4	4	3	5	1	1	1	2	4	3	2	4	3	1	0	0	0	0	1
3	3	1	1	1	2	1	4	1	1	1	1	2	1	1	2	4	2	4	3	2	2	2	4	2	1
3	2	1	1	1	2	1	5	3	2	5	1	1	1	1	4	4	2	4	3	3	2	3	4	1	1
3	2	1	3	4	2	1	4	1	2	3	1	1	1	1	2	3	2	5	1	2	0	0	0	0	1
3	8	1	1	1	2	1	3	2	2	1	1	1	3	1	4	4	3	4	2	3	2	4	3	1	1
3	2	2	1	1	3	1	3	1	2	5	1	1	1	1	3	3	2	4	4	2	4	2	3	4	1
3	8	2	1	16	3	0	5	3	1	5	1	1	1	1	5	3	3	0	3	1	3	5	1	1	1
3	2	2	1	2	2	4	3	2	1	1	1	1	1	1	2	0	2	4	0	0	0	0	0	0	1
3	5	2	1	4	2	0	2	2	2	2	1	1	1	3	0	4	0	4	0	0	0	0	0	0	1
3	2	2	1	12	2	2	5	2	3	5	1	1	1	1	3	4	2	5	1	2	2	2	3	1	1
3	2	2	1	1	2	2	4	2	2	5	1	2	1	1	5	3	2	4	1	2	3	2	1	1	1
3	2	2	1	14	2	1	5	3	2	2	1	1	1	1	2	4	2	4	1	2	3	3	4	5	1
3	2	2	2	1	2	1	2	1	3	4	1	3	2	1	2	2	2	0	3	2	3	2	0	1	1
3	8	2	1	14	4	4	3	3	1	2	1	1	1	3	2	4	2	4	3	1	5	2	3	2	1
3	3	2	1	1	2	1	3	3	3	1	1	1	1	1	4	4	2	4	4	2	0	0	0	0	1
3	2	2	2	1	4	0	3	4	4	2	1	1	1	3	3	3	3	3	3	1	2	3	0	3	1
3	3	2	1	14	0	0	5	1	1	3	1	3	1	1	5	0	0	0	0	0	0	0	0	0	1
3	3	2	1	1	2	1	3	2	1	5	1	1	1	1	4	2	2	4	3	1	4	4	2	1	1
3	5	3	1	16	3	4	5	2	2	3	1	1	1	2	4	2	5	1	2	1	1	1	0	1	1
3	8	3	0	14	2	2	5	2	3	2	1	1	1	1	2	4	3	4	1	1	2	2	1	1	1
3	8	3	1	14	3	1	5	2	2	3	1	1	1	1	2	4	2	5	1	2	0	5	0	0	1
3	5	3	1	14	3	0	3	2	2	3	1	1	1	1	3	3	2	4	2	1	2	3	1	2	1
3	2	3	3	4	2	1	5	2	3	1	1	1	1	1	3	4	3	4	1	4	3	0	0	0	1
3	8	3	1	1	3	1	5	2	3	1	1	1	1	1	1	5	5	3	1	2	1	4	1	1	1
3	2	4	1	1	2	2	4	3	4	4	1	1	1	3	2	4	3	4	3	1	4	4	1	1	1
3	2	4	1	14	3	1	5	2	1	3	1	2	1	3	2	4	0	3	3	2	3	3	3	4	1
3	8	4	1	14	2	2	2	4	3	4	1	1	1	2	3	4	4	2	1	4	3	2	1	2	1
4	2	1	1	4	3	1	3	4	4	4	1	1	1	1	2	2	3	3	2	1	0	0	0	0	1
4	8	1	1	1	2	1	2	4	1	4	1	1	1	1	2	3	3	4	2	4	4	2	1	1	1
4	4	1	4	4	2	1	3	2	3	2	1	1	1	3	0	4	3	4	0	0	0	0	0	0	1
4	2	2	0	4	3	0	2	2	3	1	1	1	3	1	1	3	2	3	0	0	0	0	0	0	1
4	8	2	1	1	3	1	5	3	4	4	1	2	0	1	2	5	3	4	1	2	3	4	1	2	1
4	8	2	1	14	1	2	5	3	2	4	1	1	1	1	2	3	2	4	3	2	2	3	1	1	1
4	3	2	1	14	3	2	2	4	4	0	1	2	2	1	4	4	3	4	2	1	0	0	0	4	1
4	2	2	1	1	3	4	3	2	2	5	1	1	1	1	2	4	2	5	3	1	4	2	1	1	1
4	3	3	1	1	3	0	2	4	2	2	1	1	1	1	1	3	2	4	1	3	1	2	4	1	1
4	8	3	1	14	2	1	5	1	1	5	1	1	1	1	3	3	2	4	3	1	3	3	3	4	1
4	7	3	1	14	3	1	4	2	2	2	1	1	1	1	3	3	2	2	1	2	2	4	1	1	1
4	2	3	1	14	3	1	3	2	2	4	1	1	1	1	4	3	2	4	1	1	2	4	1	1	1
4	3	3	1	14	2	0	5	2	2	3	1	1	1	1	3	4	2	4	2	1	0	3	1	3	1
4	7	4	3	16	3	4	3	1	2	5	1	1	2	1	5	5	3	3	2	1	2	5	1	4	1
4	8	4	1	14	3	4	5	2	2	4	1	1	3	2	2	4	2	0	1	2	5	0	3	4	1
4	8	4	1	14	2	1	5	1	1	2	1	1	1	1	1	0	2	4	4	2	0	4	0	3	1

4	8	4	1	4	3	1	5	2	3	5	1	1	1	1	0	3	3	3	0	0	0	0	0	0	1
4	3	4	1	1	3	4	2	3	2	2	1	1	1	1	2	4	4	4	2	1	2	2	4	3	1
4	7	4	1	14	2	4	5	2	1	1	1	1	1	1	2	5	3	4	5	2	1	4	1	1	1
4	4	4	0	1	4	0	3	3	1	1	1	1	1	1	3	3	2	3	2	1	0	0	0	0	1
5	8	2	3	1	3	0	2	4	4	3	1	1	1	1	2	2	5	3	1	2	3	1	4	1	1
5	3	2	1	4	2	1	2	2	1	1	1	3	1	1	1	5	5	3	0	0	0	0	0	0	1
5	4	2	3	14	2	0	5	1	1	1	1	2	2	2	1	5	5	3	1	2	1	3	3	1	1
5	2	2	4	4	3	1	3	2	2	1	1	1	1	2	3	2	2	4	0	0	0	0	0	0	1
5	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
5	2	2	0	6	2	1	2	4	3	1	1	1	3	3	3	4	0	3	0	0	0	0	0	0	1
5	2	2	1	4	2	1	5	1	2	1	1	1	1	1	2	5	2	5	1	2	0	0	0	0	1
5	2	2	2	4	1	2	5	1	1	1	1	1	1	1	3	0	0	4	0	0	0	0	0	0	1
5	8	3	1	1	2	0	2	3	3	4	1	1	1	1	2	3	3	3	1	2	4	4	1	3	1
5	4	3	1	1	3	1	3	2	4	2	1	1	1	1	4	2	2	4	5	2	1	1	4	3	1
5	8	3	1	4	3	2	4	2	3	2	1	1	1	3	1	3	0	0	5	2	0	0	0	0	1
5	3	3	4	4	3	1	5	4	3	5	1	1	1	1	4	5	3	4	0	0	0	0	0	0	1
5	2	4	1	14	2	1	3	2	2	4	1	1	1	1	0	2	3	5	1	2	2	3	1	1	1
4	4	1	4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
5	2	4	3	14	1	1	5	2	3	1	1	1	1	1	4	4	2	4	2	3	3	4	1	1	1
5	2	4	3	4	3	1	5	2	2	2	1	1	1	1	4	4	2	4	0	0	0	0	0	0	1
5	8	4	0	4	2	1	3	3	2	1	1	3	3	3	0	2	0	0	1	2	0	0	0	0	1
5	4	4	3	4	2	2	5	2	2	1	1	1	1	1	4	4	2	5	1	2	5	0	0	0	1
5	2	4	1	16	4	1	5	2	2	5	1	1	1	1	2	4	3	4	1	4	2	2	3	1	1
5	8	4	2	4	4	0	4	2	2	3	1	3	3	3	1	4	4	3	3	1	0	0	0	0	1
5	4	4	1	1	2	0	3	2	0	4	1	2	1	1	3	3	3	0	1	2	1	2	0	2	1
5	7	4	1	14	2	5	3	4	2	5	1	2	1	0	3	5	0	0	1	2	0	0	0	0	1
5	6	4	0	16	3	0	3	3	0	1	1	1	1	1	2	0	0	0	3	2	1	2	2	3	1
1	2	2	3	9	4	2	2	2	0	1	0	0	0	2	3	0	0	0	1	1	0	2	2	1	0
1	2	2	1	6	4	4	5	2	1	5	0	0	0	0	3	4	3	3	1	1	0	0	0	0	0
3	3	2	1	1	2	1	2	3	2	1	0	1	1	3	2	3	0	4	2	1	0	4	3	1	0
3	3	4	1	8	3	2	4	4	4	5	0	1	1	1	4	0	0	3	1	2	0	2	0	1	0
3	7	4	0	11	3	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	8	4	1	16	2	1	5	1	1	5	0	0	0	0	3	2	2	3	5	0	0	2	3	2	0

C.3 Summary frequency statistics

The tables presented represent the frequencies of respondents' answers. The numbers in parentheses represent the frequency of electronic and postal replies, e.g., for the four students who had < 1 year of experience in Table C.2, 3 were from the electronic survey and 1 was from the postal survey. All tables, except Tables C.1 and C.3, have one count per respondent, i.e., the sum of the total column is the number of respondents who answered the question.

	What capacity is OO used in?							
Position	Teaching		Programming		Analysis & design		Other use	
Student	3	(3,0)	32	(30,2)	19	(17,2)	0	(0,0)
Academic	24	(16,8)	21	(16,5)	11	(9,2)	4	(3,1)
S/w Eng.	14	(11,3)	117	(80,37)	77	(60,17)	1	(0,1)
Proj. Man.	4	(1,3)	26	(10,16)	24	(8,16)	3	(0,3)
Other	12	(7,5)	28	(11,17)	20	(8,12)	8	(2,6)
Total	57	(38,19)	224	(147,77)	151	(102,49)	16	(5,11)

Table C.1: Responses to Q. 1(b)

	Object-oriented experience								
Position	< 1 year		1 - 2 years		3 - 4 years		> 4 years		Total
Student	4	(3,1)	13	(12,1)	17	(16,1)	3	(3,0)	37
Academic	2	(2,0)	4	(3,1)	10	(5,5)	18	(13,5)	34
S/w Eng.	15	(3,12)	37	(22,15)	40	(30,10)	35	(30,5)	127
Proj. Man.	4	(0,4)	6	(1,5)	11	(4,7)	14	(5,9)	35
Other	0	(0,0)	9	(1,8)	7	(3,4)	24	(11,13)	40
Total	25	(8,17)	69	(39,30)	85	(58,27)	94	(62,32)	273

Table C.2: Responses to Q. 2(a)

	Object-oriented language familiarity											
Position	C++		Objective-C		Eiffel		Smalltalk		CLOS		Other	
Student	24	(22,2)	15	(14,1)	1	(1,0)	16	(16,0)	3	(3,0)	5	(4,1)
Academic	17	(14,3)	13	(9,4)	1	(1,0)	14	(7,7)	3	(3,0)	6	(4,2)
S/w Eng.	85	(67,18)	41	(26,15)	11	(9,2)	35	(23,12)	8	(8,0)	18	(5,13)
Proj. Man.	14	(7,7)	18	(5,13)	0	(0,0)	10	(3,7)	1	(1,0)	12	(0,12)
Other	19	(12,7)	11	(3,8)	4	(3,1)	18	(8,10)	0	(0,0)	8	(1,7)
Total	159	(122,37)	98	(57,41)	17	(14,3)	93	(57,36)	15	(15,0)	49	(14,35)

Table C.3: Responses to Q. 3

	Typical method size?								
Position	1-4 lines		5-10 lines		11-20 lines		> 20 lines		Total
Student	3	(3,0)	14	(14,0)	11	(10,1)	4	(3,1)	32
Academic	3	(2,1)	14	(9,5)	13	(8,5)	2	(2,0)	32
S/w Eng.	7	(6,1)	47	(26,21)	48	(36,12)	16	(13,3)	118
Proj. Man.	1	(0,1)	10	(3,7)	20	(5,15)	2	(1,1)	33
Other	6	(2,4)	16	(5,11)	12	(4,8)	2	(0,2)	36
Total	20	(13,7)	101	(57,44)	104	(63,41)	26	(19,7)	251

Table C.4: Responses to Q. 4(a)

	Method size range:										
Position	1-50 lines		1-100 lines		1-150 lines		1-200 lines		1-200+ lines		Total
Student	18	(17,1)	6	(6,0)	0	(0,0)	3	(2,1)	0	(0,0)	27
Academic	14	(8,6)	5	(4,1)	1	(1,0)	1	(0,1)	1	(1,0)	22
S/w Eng.	54	(35,19)	28	(21,7)	4	(3,1)	10	(7,3)	11	(10,1)	107
Proj. Man.	13	(2,11)	4	(2,2)	2	(1,1)	6	(1,5)	0	(0,0)	25
Other	17	(5,12)	6	(3,3)	1	(0,1)	2	(1,1)	1	(0,1)	27
Total	116	(67,49)	49	(36,13)	8	(5,3)	22	(11,11)	13	(11,2)	208

Table C.5: Responses to Q. 4(b)

	Depth of class hierarchy?								
Position	2-3 levels		4-6 levels		6 > levels		No problem		Total
Student	6	(4,2)	9	(9,0)	4	(3,1)	18	(18,0)	37
Academic	5	(4,1)	13	(10,3)	3	(2,1)	13	(7,6)	34
S/w Eng.	10	(5,5)	41	(27,14)	19	(12,7)	57	(41,16)	127
Proj. Man.	6	(1,5)	10	(2,8)	2	(1,1)	16	(6,10)	34
Other	5	(1,4)	12	(3,9)	3	(0,3)	21	(11,10)	41
Total	32	(15,17)	85	(51,34)	31	(18,13)	125	(83,42)	273

Table C.6: Responses to Q. 5

	Inheritance caused difficulty?										
Position	Never		Occasionally		Sometimes		Usually		Always		Total
Student	5	(4,1)	21	(20,1)	5	(4,1)	4	(4,0)	1	(1,0)	36
Academic	5	(3,2)	12	(8,4)	9	(6,3)	6	(5,1)	0	(0,0)	32
S/w Eng.	26	(20,6)	59	(38,21)	27	(16,11)	14	(10,4)	0	(0,0)	126
Proj. Man.	7	(4,3)	16	(4,12)	5	(1,4)	4	(0,4)	1	(0,1)	33
Other	8	(4,4)	22	(8,14)	6	(2,4)	5	(1,4)	0	(0,0)	41
Total	51	(35,16)	130	(78,52)	52	(29,23)	33	(20,13)	2	(1,1)	268

Table C.7: Responses to Q. 7

	Problems caused by naming inconsistencies?										
Position	Never		Occasionally		Sometimes		Usually		Always		Total
Student	12	(10,2)	9	(9,0)	8	(7,1)	5	(5,0)	0	(0,0)	34
Academic	7	(4,3)	7	(7,0)	10	(6,4)	5	(2,3)	2	(2,0)	31
S/w Eng.	26	(15,11)	58	(43,15)	22	(12,10)	14	(9,5)	0	(0,0)	120
Proj. Man.	11	(4,7)	12	(3,9)	7	(3,4)	4	(0,4)	0	(0,0)	34
Other	7	(2,5)	20	(9,11)	8	(2,6)	4	(2,2)	0	(0,0)	39
Total	63	(35,28)	106	(71,35)	55	(30,25)	32	(18,14)	2	(2,0)	258

Table C.8: Responses to Q. 8

	Use of local class libraries?										
Position	Never		Occasionally		Sometimes		Usually		Always		Total
Student	15	(14,1)	11	(11,0)	3	(3,0)	4	(2,2)	4	(4,0)	37
Academic	13	(8,5)	7	(4,3)	4	(3,1)	8	(7,1)	1	(1,0)	33
S/w Eng.	22	(7,15)	22	(16,6)	23	(16,7)	31	(26,5)	28	(19,9)	126
Proj. Man.	5	(1,4)	6	(1,5)	1	(0,1)	12	(5,7)	8	(2,6)	32
Other	11	(1,10)	4	(0,4)	8	(4,4)	9	(5,4)	8	(5,3)	40
Total	66	(31,35)	50	(32,18)	39	(26,13)	64	(45,19)	49	(31,18)	268

Table C.9: Responses to Q. 9

	OO: Ease of analysis and design?						
Position	Yes		No		Don't know		Total
Student	29	(27,2)	4	(3,1)	2	(2,0)	35
Academic	27	(20,7)	4	(2,2)	3	(1,2)	34
S/w Eng.	99	(72,27)	13	(5,8)	11	(4,7)	123
Proj. Man.	29	(9,20)	3	(0,3)	2	(1,1)	34
Other	35	(14,21)	3	(1,2)	3	(0,3)	41
Total	219	(142,77)	27	(11,16)	21	(8,13)	267

Table C.10: Responses to Q. 10(a)

	OO: Programmer productivity?						
Position	Yes		No		Don't know		Total
Student	27	(27,2)	2	(2,0)	6	(5,1)	35
Academic	28	(21,7)	2	(1,1)	4	(1,3)	34
S/w Eng.	102	(70,32)	15	(8,7)	8	(5,3)	125
Proj. Man.	31	(10,21)	3	(0,3)	0	(0,0)	34
Other	30	(13,17)	4	(0,4)	6	(1,5)	40
Total	218	(139,79)	26	(11,15)	24	(12,12)	268

Table C.11: Responses to Q. 10(b)

	OO: Software reuse?						
Position	Yes		No		Don't know		Total
Student	27	(24,3)	1	(1,0)	7	(7,0)	35
Academic	30	(20,10)	2	(2,0)	2	(1,1)	34
S/w Eng.	105	(69,36)	13	(10,3)	7	(4,3)	125
Proj. Man.	28	(9,19)	2	(0,2)	3	(1,2)	33
Other	35	(13,22)	2	(1,1)	4	(1,3)	41
Total	225	(135,90)	20	(14,6)	23	(14,9)	268

Table C.12: Responses to Q. 10(c)

	OO: Ease of software maintenance?						
Position	Yes		No		Don't know		Total
Student	24	(21,3)	3	(3,0)	7	(7,0)	34
Academic	24	(17,7)	3	(3,0)	7	(3,4)	34
S/w Eng.	97	(69,28)	11	(5,6)	16	(8,8)	124
Proj. Man.	29	(9,20)	3	(1,2)	2	(0,2)	34
Other	29	(11,18)	4	(1,3)	7	(3,4)	40
Total	203	(127,76)	24	(13,11)	39	(21,18)	266

Table C.13: Responses to Q. 10(d)

	Multiple inheritance useful?										
Position	Never		Occasionally		Sometimes		Usually		Always		Total
Student	3	(2,1)	10	(10,0)	8	(8,0)	8	(6,2)	4	(4,0)	33
Academic	6	(5,1)	6	(5,1)	10	(5,5)	4	(2,2)	5	(4,1)	31
S/w Eng.	8	(5,3)	37	(25,12)	29	(18,11)	28	(17,11)	12	(9,3)	114
Proj. Man.	7	(2,5)	10	(0,10)	8	(4,4)	3	(1,2)	4	(3,1)	32
Other	5	(1,4)	11	(4,7)	13	(5,8)	7	(2,5)	1	(1,0)	37
Total	29	(15,14)	74	(44,30)	68	(40,28)	50	(28,22)	26	(21,5)	247

Table C.14: Responses to Q. 11

	Continual maintenance of SP code leads to unmaintainability?										
Position	Never		Occasionally		Sometimes		Usually		Always		Total
Student	0	(0,0)	3	(3,0)	4	(4,0)	15	(13,2)	5	(4,1)	27
Academic	0	(0,0)	2	(2,0)	10	(6,4)	15	(10,5)	4	(3,1)	31
S/w Eng.	1	(1,0)	18	(11,7)	31	(19, 12)	52	(36,16)	17	(14,3)	119
Proj. Man.	0	(0,0)	2	(1,1)	13	(3,10)	12	(4,8)	6	(2,4)	33
Other	0	(0,0)	9	(3,6)	7	(3,4)	14	(6,8)	7	(1,6)	37
Total	1	(1,0)	34	(20,14)	65	(35,30)	108	(69,39)	39	(24,15)	247

Table C.15: Responses to Q. 12

	Continual maintenance of 00 code leads to unmaintainability?										
Position	Never		Occasionally		Sometimes		Usually		Always		Total
Student	1	(1,0)	16	(13,3)	8	(8,0)	3	(3,0)	0	(0,0)	28
Academic	0	(0,0)	17	(12,5)	9	(8,1)	3	(3,0)	1	(1,0)	30
S/w Eng.	2	(2,0)	51	(31,20)	48	(36,12)	4	(2,2)	8	(6,2)	113
Proj. Man.	2	(2,0)	14	(3,11)	15	(4,11)	2	(1,1)	1	(0,1)	34
Other	0	(0,0)	17	(7,10)	10	(4,6)	2	(1,1)	3	(0,3)	32
Total	5	(5,0)	115	(66,49)	90	(57,33)	14	(9,5)	13	(7,6)	237

Table C.16: Responses to Q. 13

	OO code more maintainable than SP code?										
Position	Never		Occasionally		Sometimes		Usually		Always		Total
Student	0	(0,0)	0	(0,0)	3	(3,0)	18	(17,1)	8	(6,2)	29
Academic	0	(0,0)	1	(1,0)	3	(2,1)	21	(14,7)	8	(6,2)	33
S/w Eng.	2	(0,2)	5	(4,1)	22	(13,9)	69	(48,21)	22	(18,4)	120
Proj. Man.	2	(1,1)	1	(0,1)	8	(2,6)	18	(4,14)	4	(3,1)	33
Other	0	(0,0)	1	(0,1)	10	(4,6)	20	(10,10)	4	(0,4)	35
Total	4	(1,3)	8	(5,3)	46	(24,22)	146	(93,53)	46	(33,13)	250

Table C.17: Responses to Q. 14

	C++ as a de facto standard?										
Position	Bad		Good		Indifferent		Don't know		Disagree		Total
Student	25	(23,2)	5	(4,1)	1	(1,0)	1	(1,0)	0	(0,0)	32
Academic	29	(20,9)	0	(0,0)	1	(0,1)	0	(0,0)	1	(1,0)	31
S/w Eng.	54	(40,14)	21	(15,6)	25	(13,12)	3	(1,2)	6	(5,1)	109
Proj. Man.	12	(6,6)	11	(3,8)	4	(0,4)	1	(0,1)	1	(0,1)	29
Other	23	(11,12)	5	(2,3)	4	(2,2)	0	(0,0)	2	(0,2)	34
Total	143	(100,43)	42	(24,18)	35	(16,19)	5	(2,3)	10	(6,4)	235

Table C.18: Responses to Q. 15

	C++ allows hybrid programming?								
Position	Advantage		Disadvantage		Don't know		Adv. & Disadv.		Total
Student	10	(9,1)	12	(10,2)	6	(6,0)	3	(3,0)	31
Academic	7	(4,3)	21	(15,6)	2	(1,1)	1	(1,0)	31
S/w Eng.	46	(33,13)	45	(28,17)	6	(4,2)	14	(11,3)	111
Proj. Man.	14	(4,10)	10	(3,7)	2	(0,2)	3	(2,1)	29
Other	9	(6,3)	20	(7,13)	1	(0,1)	4	(2,2)	34
Total	86	(56,30)	108	(63,45)	17	(11,6)	25	(19,6)	236

Table C.19: Responses to Q. 16

	Use of Friend function rather than redesign?										
Position	Never		Occasionally		Sometimes		Usually		Always		Total
Student	3	(3,0)	5	(5,0)	7	(7,0)	6	(5,1)	0	(0,0)	21
Academic	12	(8,4)	1	(0,1)	2	(2,0)	4	(3,0)	0	(0,0)	19
S/w Eng.	20	(16,4)	41	(30,11)	21	(12,9)	10	(6,4)	3	(2,1)	95
Proj. Man.	5	(2,3)	11	(5,6)	3	(1,2)	2	(0,2)	2	(0,2)	23
Other	5	(0,5)	11	(7,4)	4	(2,2)	2	(0,2)	1	(0,1)	23
Total	45	(29,16)	69	(47,22)	37	(24,13)	24	(14,10)	6	(2,4)	181

Table C.20: Responses to Q. 17

	Make use of operator overloading?										
Position	Never		Occasionally		Sometimes		Usually		Always		Total
Student	4	(4,0)	7	(6,1)	2	(2,0)	6	(6,0)	4	(4,0)	23
Academic	9	(8,1)	3	(3,0)	5	(3,2)	4	(1,3)	2	(0,2)	23
S/w Eng.	12	(8,4)	36	(24,12)	18	(13,5)	20	(16,4)	9	(6,3)	95
Proj. Man.	1	(1,0)	7	(2,5)	7	(4,3)	6	(0,6)	3	(1,2)	24
Other	3	(1,2)	6	(2,4)	7	(4,3)	5	(2,3)	1	(1,0)	21
Total	29	(22,7)	59	(37,22)	39	(26,13)	41	(25,16)	19	(12,7)	187

Table C.21: Responses to Q. 18(a)

	Overload operators as:								
Position	Members		Non-members		Both		N/A		Total
Student	7	(7,0)	3	(3,0)	5	(5,0)	4	(4,0)	19
Academic	7	(3,4)	0	(0,0)	4	(2,2)	10	(9,1)	21
S/w Eng.	29	(19,10)	6	(4,2)	26	(21,5)	19	(12,7)	80
Proj. Man.	11	(1,10)	0	(0,0)	6	(3,3)	4	(2,2)	21
Other	8	(4,4)	1	(0,1)	9	(5,4)	3	(1,2)	21
Total	62	(34,28)	10	(7,3)	50	(36,14)	40	(28,12)	162

Table C.22: Responses to Q. 18(b)

	Make use of templates?										
Position	Never		Occasionally		Sometimes		Usually		Always		Total
Student	10	(9,1)	3	(3,0)	0	(0,0)	8	(8,0)	2	(2,0)	23
Academic	7	(2,5)	6	(6,0)	6	(4,2)	1	(1,0)	2	(1,1)	22
S/w Eng.	48	(34,12)	17	(13,4)	11	(9,2)	12	(7,5)	4	(2,2)	92
Proj. Man.	9	(2,7)	4	(2,2)	7	(2,5)	6	(2,4)	0	(0,0)	26
Other	9	(4,5)	2	(0,2)	6	(2,4)	5	(4,1)	0	(0,0)	22
Total	83	(51,32)	32	(24,8)	30	(17,13)	32	(22,10)	8	(5,3)	185

Table C.23: Responses to Q. 19

Appendix D

Experimental Materials And Collected Data

D.1 Instructions for the first experiment

D.1.1 Practical Test Instructions

Please read these instructions carefully. Any questions of clarification will be answered by the invigilators. The test is open-book, i.e., you can refer to class notes and the recommended textbook (but only the recommended textbook). This test contributes 30% of the overall assessment. Marks will be awarded for correctness. On completing the test, candidates may leave the laboratory. Candidates may not otherwise leave the laboratory until the time is up. If there is an emergency, such as a workstation crash, please signal one of the invigilators.

- Check that you have two packets, numbered 1 and 2. If you do not have these, indicate this to an invigilator.
- **When instructed,** open Packet 1 which contains information about a program and a modification you are asked to make to it. You have 10 minutes to read this information, and may ask questions for clarification during this time. Please ensure that you understand what is required.
- **When instructed,** open Packet 2 which contains the listing of the source code you will be modifying. Approach the modification in the way you see fit. Please make sure you work as carefully as you can.

- Anything you wish to write down should be done so on the source code listing. In the event of being unable to complete the test, documentary evidence of your work will attract marks.
- Make sure you print and sign your name on the listing of the source code.
- Once you feel you have completed the modification specified in Packet 1, indicate this to an invigilator who will examine your program output. If your modification is correct and produces the desired output then you have completed the test, otherwise you must continue making the modification until you either successfully complete the task, or until the test time is up. You will not be penalised if you have to continue after showing the results of your modification to an invigilator.
- When you have successfully completed the test, an invigilator will collect your annotated listing of the source code and sign it if your work was successfully demonstrated.
- After the test is over, on no account alter the files you have used in this practical test. These will be printed later, used as documentary evidence of your work, and will attract marks.

To avoid candidates being in breach of University regulations regarding conduct at examinations, the invigilators insist that throughout the test:

- there is no talking
- there is no use of the printers
- there is no use of e-mail or any other form of electronic communication
- no attempt is made to look at another candidate's work

D.1.2 General Information: Test 1

Program functionality: the program creates objects from the subset of a university population, namely student, lecturer and secretary. It then manipulates these objects through some of their member functions.

Each member of the university community has a set of particular attributes, some of which are common to all members. The attributes for each member are as follows:

Student	Lecturer	Secretary
last_name	last_name	last_name
first_name	first_name	first_name
age	age	age
department	department	department
registration_number	staff_id_number	staff_id_number
year	annual_salary	hourly_wage
		boss

Modification Overview

Program extensibility: you are required to extend the program to deal with a new member, professor, of the university community, ie write a class professor, and create an instance of this class to show your code performs correctly. See the Detailed Specification section for a complete description of this modification.

File structure

The file structure consists of a header and implementation file for each class, and a main file. These files are conventionally named: that is header files are filename.h, implementation files are filename.cc, and the main file is main.cc.

Furthermore, there exist files in your directory which, while they must present for the compilation to work, **do not need to be modified or even examined**. These files are: string.h and string.cc (they simply provide the means to store a sequence of characters as a string object), and compile (used to compile the program - see next section).

Compiling and Executing the Program

For examination purposes a shell script has been introduced which saves your files each time you compile your code. This enables examiners to award marks for incomplete work. You compile your program by typing:

```
compile *.cc
```

Note: compile is used in place of CC.

You must then deal with any compilation errors you may receive. Once your program compiles, you may execute it by typing:

```
a.out
```

Detailed Specification

1. Write a class **professor** which has the following attributes:

String last_name;	String first_name;
unsigned age;	String department;
unsigned staff_id_number;	float annual_salary;
integer research_grant_number;.	

   Your professor class should be able to construct an object using parameters, and should be able to set all of its attributes when required, just like any of the other classes. Furthermore, when a professor object is displayed to the screen it should display all its private data in a fashion similar to other objects. (see Program Output section for clarification of this output).

   You must write your code as a header file called professor.h, and an implementation file called professor.cc. Once you have written the appropriate code, add the lines

   ```
   professor Williams("Williams", "John", 53, "History",
                      100003, 31000, 0);
   Williams.print();
   Williams.set_research_grant_number(13);
   Williams.print();
   ```

   to function main, in file main.cc. This will create your professor object, display it, use the member function which sets the research_grant_number, and then display the object again to make sure the desired effect has been achieved.

Program Output

You may wish to compile the program before you make any modifications and examine the program output. A lecturer object is output, his department is changed from Mathematics to Statistics, and then the lecturer object is output again.

The actual program output **after** a correct modification should be:

```
Name: Smith, David
Age: 34
Department: Mathematics
Staff Id. #: 100001
Annual Salary: $25000

Name: Smith, David
Age: 34
Department: Statistics
Staff Id. #: 100001
Annual Salary: $25000

Name: Williams, John
Age: 53
Department: History
Staff Id. #: 100003
Annual Salary: $31000
Research Grant #: 0

Name: Williams, John
Age: 53
Department: History
Staff Id. #: 100003
Annual Salary: $31000
Research Grant #: 13
```

If you have a problem viewing the program output because it scrolls off the screen, you can execute the program by typing

```
a.out | more
```

When you feel you have completed the test satisfactorily, raise your hand and demonstrate your output to an invigilator.

Getting Started

Login to the workstation and start up the X windowing system by typing `startx`. The source code for your practical test has been copied into a subdirectory called C++/Test1. Type `cd C++/Test1` to enter the directory, followed by `ls` to list the files. You can examine any file by (i) looking at the listing in Packet 2, (ii) by typing `more filename` or (iii) loading the file into the `emacs` editor. You should run the `emacs` editor from one window and use a separate window for issuing compilation commands and execution commands. Make sure to `cd C++/Test1` in both windows.

Important: before you open Packet 2, it is imperative that you script your entire practical session (that is, make a record of your work on the computer). In the window that you plan to use for compilation and execution commands, type:

```
record
```

This will start a script file called typescript which records your workstation session, and changes your prompt to look something like

```
maxwell-04 jd 1:55pm %
```

If this does not happen when you type `record` please raise your hand and tell an invigilator. It is important that you get this working as it enables the examiner to determine how serious an attempt at modifying, compiling and executing the program has been made. As stressed earlier, documentary evidence of your work will attract marks.

Important: At the end of the test, please type

```
exit
```

to finish the auto recording of your programming efforts.

You should now understand what is required of you, and be ready to open Packet 2.

D.1.3 General Information: Test 2 (and Internal Replication)

Program functionality: the program creates objects from the subset of sources of written information, namely book, conference, in_conference (a paper from a conference proceedings), and thesis. It then manipulates these objects through some of their member functions.

Each member of the written work subset has a set of particular attributes, some of which are common to all members. The attributes for each member are as follows:

Book	**Conference**	**In_conference**	**Thesis**
citekey	citekey	citekey	citekey
title	title	title	title
year	year	year	year
author	editor	editor	author
publisher	publisher	publisher	institution
volume	organisation	organisation	
		paper_title	
		author	
		pages	

Modification Overview

Program extensibility: you are required to extend the program to deal with a new member, phd thesis, of the written information subset, ie write a new class phd_thesis, and create an instance of this class to show your code performs correctly. See the Detailed Specification section for a complete description of this modification.

File structure

The file structure consists of a header file and implementation file for each class, and a main file. These files are conventionally named: that is header files are filename.h, implementation files are filename.cc, and the main file is main.cc.

Furthermore, there exist files in your directory which, while they must present for the compilation to work, **do not need to be modified or even examined.** These files are: string.h and string.cc (they simply provide the means to store a sequence of characters as a string object), and compile (used to compile the program - see next section).

Compiling and Executing the Program

For examination purposes a shell script has been introduced which saves your files each time you compile your code. This enables examiners to award marks for incomplete work. You compile your program by typing:

```
compile *.cc
```

Note: compile is used in place of CC.

You must then deal with any compilation errors you receive. Once your program compiles, you may execute it simply by typing:

```
a.out
```

Detailed Specification

1. Write a class **phd_thesis** which has the following attributes:

String citekey;	String title;
unsigned year;	String author;
String institution;	String supervisor;
unsigned duration;	

   Your phd_thesis class should be able to construct an abject using parameters, and should be able to set all of its attributes when required, just like any of the other classes. Furthermore, when the phd_thesis object is displayed to the screen it should display its private data in a fashion similar to other objects (see Program Output section for clarification of this output).

   You must write your code as a header file called phd_thesis.h, and an implementation file called phd_thesis.cc. Once you have written the appropriate code, add the lines

   ```
   phd_thesis Johnson("johnson:85",
    "Software Understandability: An Empirical Study",
    1985, "D. Johnson", "Georgia State University", "unknown", 0);
   Johnson.print();
   Johnson.set_supervisor("Dr. M. Watson");
   Johnson.set_duration(3);
   Johnson.print();
   ```

   to the function main, in file main.cc. This will create your phd_thesis object, display it, use the member functions which set the duration and supervisor, and then display the object again to make sure the desired effect has been achieved.

Program Output

You may wish to compile the program before you make any modifications and examine the program output. A conference object is output, the editor is changed from a missing value to W. Schaefor, and then the conference object is output again.

The actual program output **after** a correct modification should be:

```
cite key: ieee:87
Title: IEEE Tutorial on Software Design Techniques
Year: 1987
Editor:
Publisher: McGraw-Hill
Organisation: IEEE

cite key: ieee:87
Title: IEEE Tutorial on Software Design Techniques
Year: 1987
Editor: W. Schaefor
Publisher: McGraw-Hill
Organisation: IEEE

cite key: johnson:85
Title: Software Understandability: An Empirical Study
Year: 1985
Author: D. Johnson
Academic Institution: Georgia State University
Supervisor: unknown
Course Duration (years): 0

cite key: johnson:85
Title: Software Understandability: An Empirical Study
Year: 1985
Author: D. Johnson
Academic Institution: Georgia State University
Supervisor: Dr. M. Watson
Course Duration (years): 3
```

If you have a problem viewing the program output because it scrolls off the screen, you can execute the program by typing

```
a.out | more
```

When you feel you have completed the test satisfactorily, raise your hand and demonstrate your output to an invigilator.

Getting Started

Login to the workstation and start up the X windowing system by typing `startx`. The source code for your practical test has been copied into a subdirectory called C++/Test2. Type `cd C++/Test2` to enter the directory, followed by `ls` to list the files. You can examine any file by (i) looking at the listing in Packet 2, (ii) by typing `more filename` or (iii) loading the file into the `emacs` editor. You should run the `emacs` editor from one window and use a separate window for issuing compilation commands and execution commands. Make sure to `cd C++/Test2` in both windows.

Important: before you open Packet 2, it is imperative that you script your entire practical session (that is, make a record of your work on the computer). In the window that you plan to use for compilation and execution commands, type:

```
record
```

This will start a script file called typescript which records your workstation session, and changes your prompt to look something like

```
maxwell-04 jd 1:55pm %
```

If this does not happen when you type `record` please raise your hand and tell an invigilator. It is important that you get this working as it enables the examiner to determine how serious an attempt at modifying, compiling and executing the program has been made. As stressed earlier, documentary evidence of your work will attract marks.

Important: At the end of the test, please type

```
exit
```

to finish the auto recording of your programming efforts.

You should now understand what is required of you, and be ready to open Packet 2.

D.2 Code for first experiment

D.2.1 Test 1

Inheritance program version

```
// lecturer.h
class lecturer : public staff
{
 private:
     float annual_salary;

 public:
     /* lecturer constructor */
     lecturer (String, String, int, String, int, float);

     void set_annual_salary (float);
     /* allow print to be overwritten by derived classes: make it virtual */
     virtual void print () const;
};

// lecturer.cc
#include <iostream.h>
#include "univ_community.h"
#include "staff.h"
#include "lecturer.h"

// assign initial values
lecturer::lecturer (String ln, String fn, int a, String dept, int sin, float s) :
                    staff (ln, fn, a, dept, sin)
{
     annual_salary = s;
}

void lecturer::set_annual_salary (float as)
{
     annual_salary = as;
}

void lecturer::print () const
{
     staff::print();
     cout << "Annual Salary: $" << annual_salary << endl;
}

// secretary.h
class secretary : public staff
{
 private:
     float hourly_wage;
     String boss;

 public:
     /* secretary constructor */
     secretary (String, String, int, String, int, float, String);

     void set_hourly_wage (float);
     void set_boss (String);
     /* allow print to be overwritten by derived classes: make it virtual */
     virtual void print () const;
};

// secretary.cc
#include <iostream.h>
```

```
#include "univ_community.h"
#include "staff.h"
#include "secretary.h"

// assign initial values
secretary::secretary (String ln, String fn, int a, String dept, int sin,
                      float hw, String bs) :
                      staff (ln, fn, a, dept, sin)
{
     hourly_wage = hw;
     boss = bs;
}

void secretary::set_hourly_wage (float hw)
{
     hourly_wage = hw;
}

void secretary::set_boss (String bs)
{
     boss = bs;
}

void secretary::print () const
{
     staff::print();
     cout << "Hourly Wage: $" << hourly_wage << endl
          << "Boss: "<< boss << endl;
}

// staff.h
class staff : public univ_community
{
 private:
     int staff_id_number;

 public:
     /* staff constructor */
     staff (String, String, int, String, int);

     void set_staff_id_number (int);
     /* allow print to be overwritten by derived classes: make it virtual */
     virtual void print () const;
};

// staff.cc
#include <iostream.h>
#include "univ_community.h"
#include "staff.h"

// assign initial values
staff::staff (String ln, String fn, int a, String dept, int sin) :
univ_community (ln, fn, a, dept)
{
     staff_id_number = sin;
}

void staff::set_staff_id_number(int sin)
{
     staff_id_number = sin;
}

void staff::print () const
{
     univ_community::print ();
```

```
     cout << "Staff Id. #: " << staff_id_number << endl;
}

// student.h
class student : public univ_community
{
 private:
     int registration_number;
     int year;

 public:
     /* student constructor */
     student (String, String, int, String, int, int);

     void set_registration_number (int);
     void set_year (int);
     /* allow print to be overwritten by derived classes: make it virtual  */
     virtual void print () const;
};

// student.cc
#include <iostream.h>
#include "univ_community.h"
#include "student.h"

// assign initial values
student::student (String ln, String fn, int a, String dept, int reg_numb, int yr) :
univ_community (ln, fn, a, dept)
{
     registration_number = reg_numb;
     year = yr;
}

void student::set_registration_number (int reg_numb)
{
     registration_number = reg_numb;
}

void student::set_year (int yr)
{
     year = yr;
}

void student::print () const
{
     univ_community::print();
     cout << "Registration #: " << registration_number << endl
          << "Year: " << year << endl;
}

// univ_community.h
#include "string.h"    /* Header file for the String Class */

class univ_community
{
 private:
     String last_name; /* String object which stores last name */
     String first_name;
     int age;
     String department;

 public:
     /* univ_community constructor */
     univ_community (String, String, int, String);
```

```
     void set_last_name (String);
     void set_first_name (String);
     void set_age (int);
     void set_department (String);
     /* allow print to be overwritten by derived classes: make it virtual */
     virtual void print () const;
};

// univ_community.cc
#include <iostream.h>
#include "univ_community.h"

// assign initial values
univ_community::univ_community (String ln, String fn, int a, String dept)
{
     last_name = ln;
     first_name = fn;
     age = a;
     department = dept;
}

void univ_community::set_last_name (String last_n)
{
     last_name = last_n;
}

void univ_community::set_first_name (String first_n)
{
     first_name = first_n;
}

void univ_community::set_age (int a)
{
     age = a;
}

void univ_community::set_department(String dept)
{
     department = dept;
}

void univ_community::print () const
{
     cout << endl << "Name: " << last_name << ", "
          << first_name << endl
          << "Age: " << age << endl
          << "Department: " << department << endl;
}

// main.cc
#include "univ_community.h"     // Header file for Univ_community Class
#include "student.h"            // Header file for Student Class
#include "staff.h"              // Header file for Staff Class
#include "lecturer.h"           // Header file for Lecturer Class
#include "secretary.h"          // Header file for Secretary Class

main()
{
     /* create objects and assign values using the constructor */
     student Daly("Daly", "John", 22, "Computer Science", 9263520, 2);
     student Watson("Watson", "Andy", 26, "Economics", 9164789, 4);
     lecturer Smith("Smith", "David", 34, "Mathematics", 100001, 25000);
     secretary Jones("Jones", "Anne", 0, "English", 0, 0, "GTB");

     /* change the private data using the object's member functions */
```

```
     Jones.set_age(21);
     Jones.set_staff_id_number(100002);
     Jones.set_hourly_wage(5.55);

     /* print the Smith object, change a private data member, and */
     /* observe the outcome */
     Smith.print();
     Smith.set_department("Statistics");
     Smith.print();
}
```

Flat program version

```
// lecturer.h
#include "string.h"     /* Header file for the String class */

class lecturer
{
 private:
     String last_name;  /* String object which stores last name */
     String first_name;
     int age;
     String department;
     int staff_id_number;
     float annual_salary;

 public:
     /* lecturer constructor */
     lecturer (String, String, int, String, int, float);

     void set_last_name (String);
     void set_first_name (String);
     void set_age (int);
     void set_department (String);
     void set_staff_id_number (int);
     void set_annual_salary (float);
     void print () const;
};

// lecturer.cc
#include <iostream.h>
#include "lecturer.h"

// assign initial values
lecturer::lecturer (String ln, String fn, int a, String dept, int sin, float s)
{
     last_name = ln;
     first_name = fn;
     age = a;
     department = dept;
     staff_id_number = sin;
     annual_salary = s;
}

void lecturer::set_last_name (String last_n)
{
     last_name = last_n;
}

void lecturer::set_first_name (String first_n)
{
     first_name = first_n;
}
void lecturer::set_age (int a)
{
```

```
     age = a;
}

void lecturer::set_department(String dept)
{
     department = dept;
}

void lecturer::set_staff_id_number(int sin)
{
     staff_id_number = sin;
}

void lecturer::set_annual_salary (float as)
{
     annual_salary = as;
}

void lecturer::print () const
{
     cout << endl << "Name: " << last_name << ", "
          << first_name << endl
          << "Age: " << age << endl
          << "Department: " << department << endl
          << "Staff Id. #: " << staff_id_number << endl
          << "Annual Salary: $" << annual_salary << endl;
}

// secretary.h
#include "string.h"     /* Header file for the String Class */

class secretary
{
 private:
     String last_name;  /* String object which stores the last name */
     String first_name;
     int age;
     String department;
     int staff_id_number;
     float hourly_wage;
     String boss;

 public:
     /* secretary constructor */
     secretary (String, String, int, String, int, float, String);

     void set_last_name (String);
     void set_first_name (String);
     void set_age (int);
     void set_department (String);
     void set_staff_id_number (int);
     void set_hourly_wage (float) ;
     void set_boss (String);
     void print () const;
};

// secretary.cc
#include <iostream.h>
#include "secretary.h"

// assign initial values
secretary::secretary (String ln, String fn, int a,
                      String dept, int sin, float hw, String bs)
{
     last_name = ln;
```

```
     first_name = fn;
     age = a;
     department = dept;
     staff_id_number = sin;
     hourly_wage = hw;
     boss = bs;
}

void secretary::set_last_name (String last_n)
{
     last_name = last_n;
}

void secretary::set_first_name (String first_n)
{
     first_name = first_n;
}

void secretary::set_age (int a)
{
     age = a;
}

void secretary::set_department(String dept)
{
     department = dept;
}

void secretary::set_staff_id_number(int sin)
{
     staff_id_number = sin;
}

void secretary::set_hourly_wage (float hw)
{
     hourly_wage = hw;
}

void secretary::set_boss (String bs)
{
     boss = bs;
}

void secretary::print () const
{
     cout << endl << "Name: " << last_name << ", "
          << first_name << endl
          << "Age: " << age << endl
          << "Department: " << department << endl
          << "Staff Id. #: " << staff_id_number << endl
          << "Hourly Wage: $" << hourly_wage << endl
          << "Boss: "<< boss << endl;
}

// student.h
#include "string.h"     /* Header file for the String Class */

class student
{
 private:
     String last_name;  /* String object which stores the last name */
     String first_name;
     int age;
     String department;
     int registration_number;
```

```
     int year;

 public:
     /* student constructor */
     student (String, String, int, String, int, int);

     void set_last_name (String);
     void set_first_name (String);
     void set_age (int);
     void set_department (String);
     void set_registration_number (int);
     void set_year (int);
     void print () const;
};

// student.cc
#include <iostream.h>
#include "student.h"

// assign initial values
student::student (String ln, String fn, int a, String dept,
                  int reg_numb, int yr)
{
     last_name = ln;
     first_name = fn;
     age = a;
     department = dept;
     registration_number = reg_numb;
     year = yr;
}

void student::set_last_name (String last_n)
{
     last_name = last_n;
}

void student::set_first_name (String first_n)
{
     first_name = first_n;
}

void student::set_age (int a)
{
     age = a;
}

void student::set_department(String dept)
{
     department = dept;
}

void student::set_registration_number (int reg_numb)
{
     registration_number = reg_numb;
}

void student::set_year (int yr)
{
     year = yr;
}

void student::print () const
{
     cout << endl << "Name: " << last_name << ", "
          << first_name << endl
```

```
          << "Age: " << age << endl
          << "Department: " << department << endl
          << "Registration Number: " << registration_number << endl
          << "Year: " << year << endl;
}

// main.cc
Same as inheritance program version
```

D.2.2 Test 2 (and Internal Replication)

Inheritance program version

```
// book.h
class book : public written_work
{
 private:
     String author;
     String publisher;
     int volume;

 public:
     /* book constructor */
     book (String, String, int, String, String, int);

     void set_author (String);
     void set_publisher (String);
     void set_volume (int);
     void print () const;
};

// book.cc
#include <iostream.h>
#include "written_work.h"
#include "book.h"

// assign initial values
book::book (String ck, String tle, int yr, String athr, String pub, int vol) :
written_work (ck, tle, yr)
{
     author = athr;
     publisher = pub;
     volume = vol;
}

void book::set_author (String athr)
{
     author = athr;
}

void book::set_publisher (String pub)
{
     publisher = pub;
}

void book::set_volume (int vol)
{
     volume = vol;
}

void book::print () const
{
     written_work::print();
     cout << "Author: " << author << endl
          << "Publisher: " << publisher << endl
```

```
          << "Volume: " << volume << endl;
}

// conference.h
class conference : public written_work
{
 private:
     String editor;
     String publisher;
     String organisation;

 public:
     /* conference constructor */
     conference (String, String, int, String, String, String) ;

     void set_editor (String);
     void set_publisher (String);
     void set_organisation (String);
     void print () const;
};

// conference.cc
#include <iostream.h>
#include "written_work.h"
#include "conference.h"

// assign initial values
conference::conference (String ck, String tle, int yr, String ed, String pub,
                        String org) :
                        written_work (ck, tle, yr)
{
     editor = ed;
     publisher = pub;
     organisation = org;
}

void conference::set_editor (String ed)
{
     editor = ed;
}

void conference::set_publisher (String pub)
{
     publisher = pub;
}

void conference::set_organisation (String org)
{
     organisation = org;
}

void conference::print () const
{
     written_work::print();
     cout << "Editor: " << editor << endl
          << "Publisher: " << publisher << endl
          << "Organisation: " << organisation << endl;
}

// in_conference.h
class in_conference : public conference
{
 private:
     String paper_title;
     String author;
```

```
     String pages;

 public:
     /* in_conference constructor */
     in_conference (String, String, int, String, String, String, String,
                    String, String);

     void set_paper_title (String);
     void set_author (String);
     void set_pages (String);
     void print () const;
};

// in_conference.cc
#include <iostream.h>
#include "written_work.h"
#include "conference.h"
#include "in_conference.h"

// assign initial values
in_conference::in_conference (String ck, String tle, int yr, String ed,
                              String pub, String org, String pt, String aut,
                              String pgs) :
                              conference (ck, tle, yr, ed, pub, org)
{
     paper_title = pt;
     author = aut;
     pages = pgs;
}

void in_conference::set_paper_title (String pt)
{
     paper_title = pt;
}

void in_conference::set_author (String aut)
{
     author = aut;
}

void in_conference::set_pages (String pgs)
{
     pages = pgs;
}

void in_conference::print () const
{
     conference::print();
     cout << "Paper Title: " << paper_title << endl
          << "Author: " << author << endl
          << "Pages: " << pages << endl;
}

// report.h
class report : public written_work
{
 private:
     String author;

 public:
     /* report constructor */
     report (String, String, int, String);

     void set_author (String);
     void print () const;
```

```
};

// report.cc
#include <iostream.h>
#include "written_work.h"
#include "report.h"

// assign initial values
report::report (String ck, String tle, int yr, String aut) :
written_work (ck, tle, yr)
{
     author = aut;
}

void report::set_author (String aut)
{
     author = aut;
}

void report::print () const
{
     written_work::print();
     cout << "Author: " << author << endl;
}

// thesis.h
class thesis : public report
{
 private:
     String institution;

 public:
     /* thesis constructor */
     thesis (String, String, int, String, String);

     void set_institution (String);
     void print () const;
};

// thesis.cc
#include <iostream.h>
#include "written_work.h"
#include "report.h"
#include "thesis.h"

// assign initial values
thesis::thesis (String ck, String tle, int yr, String aut, String inst) :
report (ck, tle, yr, aut)
{
     institution = inst;
}

void thesis::set_institution (String inst)
{
     institution = inst;
}

void thesis::print () const
{
     report::print();
     cout << "Academic Institution: " << institution << endl;
}

// written_work.h
#include "string.h"     /* Header file for the String Class */
```

```
class written_work
{
 private:
     String citekey;    /* String object used to hold the cite key value */
     String title;
     int year;

 public:
     /* written_work constructor */
     written_work (String, String, int);

     void set_citekey (String);
     void set_title (String);
     void set_year (int);
     /* allow print to be overwritten by derived classes: make it virtual */
     virtual void print () const;
};

// written_work.cc
#include <iostream.h>
#include "written_work.h"

// assign initial values
written_work::written_work (String ck, String tle, int yr)
{
     citekey = ck;
     title = tle;
     year = yr;
}

void written_work::set_citekey (String ck)
{
     citekey = ck;
}

void written_work::set_title (String tle)
{
     title = tle;
}

void written_work::set_year (int yr)
{
     year = yr;
}

void written_work::print () const
{
     cout << endl << "cite key: " << citekey << endl
          << "Title: " << title << endl
          << "Year: "  << year << endl;
}

// main.cc
#include "written_work.h"       // header file for written_work class
#include "book.h"         // header file for book class
#include "conference.h"         // header file for conference class
#include "in_conference.h"      // header file for in_conference class
#include "report.h"         // header file for report class
#include "thesis.h"             // header file for thesis class

main()
{
     /* create the objects and assign values using the constructor */
     book stats("cohen:82", "Introductory Statistics", 1982, "J. Cohen",
```

```
        "Academic Press", 1);

     in_conference oop("minor:92", "The Int. Conf. on Software Maintenance",
                       1992, "E. Smith", "Ablex & Co.", "IEEE",
                       "The OO Paradigm for Software Maintenance", "E. Minor",
                       "81 - 82");

     thesis domains("daly:91", "Semantic Integration of Heterogeneous Domains",
                    1991, "J Daly", "Strathclyde University");

     conference ieee("ieee:87", "IEEE Tutorial on Software Design Techniques",
                     0, "", "", "IEEE");

     /* set the private data using the object's member functions */
     ieee.set_year(1987);
     ieee.set_publisher("McGraw-Hill");

     /* print the ieee object, set a private data member, and */
     /* observe the outcome */
     ieee.print();
     ieee.set_editor("W. Schaefor");
     ieee.print();
}
```

Flat program version

```
// book.h
#include "string.h"

class book
{
 private:
     String citekey;    /* String object used to hold the cite key value  */
     String title;
     int year;
     String author;
     String publisher;
     int volume;

 public:
     /* book constructor */
     book (String, String, int, String, String, int);

     void set_citekey (String);
     void set_title (String);
     void set_year (int);
     void set_author (String);
     void set_publisher (String);
     void set_volume (int);
     void print () const;
};

// book.cc
#include <iostream.h>
#include "book.h"

// assign initial values
book::book (String ck, String tle, int yr, String athr, String pub, int vol)
{
     citekey = ck;
     title = tle;
     year = yr;
     author = athr;
     publisher = pub;
     volume = vol;
```

```
}

void book::set_citekey (String ck)
{
     citekey = ck;
}
void book::set_title (String tle)
{
     title = tle;
}

void book::set_year (int yr)
{
     year = yr;
}

void book::set_author (String athr)
{
     author = athr;
}

void book::set_publisher (String pub)
{
     publisher = pub;
}

void book::set_volume (int vol)
{
     volume = vol;
}

void book::print () const
{
     cout << endl << "cite key: " << citekey << endl
          << "Title: " << title << endl
          << "Year: "  << year << endl
          << "Author: " << author << endl
          << "Publisher: " << publisher << endl
          << "Volume: " << volume << endl;
}

// conference.h
#include "string.h"     /* Header file for the String Class */

class conference
{
 private:
     String citekey;
     String title;
     int year;
     String editor;
     String publisher;
     String organisation;

 public:
     /* conference constructor */
     conference (String, String, int, String, String, String) ;

     void set_citekey (String);
     void set_title (String);
     void set_year (int);
     void set_editor (String);
     void set_publisher (String);
     void set_organisation (String);
     void print () const;
```

```
};

// conference.cc
#include <iostream.h>
#include "conference.h"

// assign initial values
conference::conference (String ck, String tle, int yr,
                        String ed, String pub, String org)
{
     citekey = ck;
     title = tle;
     year = yr;
     editor = ed;
     publisher = pub;
     organisation = org;
}

void conference::set_citekey (String ck)
{
     citekey = ck;
}

void conference::set_title (String tle)
{
     title = tle;
}

void conference::set_year (int yr)
{
     year = yr;
}

void conference::set_editor (String ed)
{
     editor = ed;
}

void conference::set_publisher (String pub)
{
     publisher = pub;
}

void conference::set_organisation (String org)
{
     organisation = org;
}

void conference::print () const
{
     cout << endl << "cite key: " << citekey << endl
          << "Title: " << title << endl
          << "Year: "  << year << endl
          << "Editor: " << editor << endl
          << "Publisher: " << publisher << endl
          << "Organisation: " << organisation << endl;
}

// in_conference.h
#include "string.h"     /* Header file for the String Class */

class in_conference
{
 private:
     String citekey;
```

```
    String title;
    int year;
    String editor;
    String publisher;
    String organisation;
    String paper_title;
    String author;
    String pages;

 public:
    /* in_conference constructor */
    in_conference (String, String, int, String, String, String, String,
                   String, String);

    void set_citekey (String);
    void set_title (String);
    void set_year (int);
    void set_editor (String);
    void set_publisher (String);
    void set_organisation (String);
    void set_paper_title (String);
    void set_author (String);
    void set_pages (String);
    void print () const;
};

// in_conference.cc
#include <iostream.h>
#include "in_conference.h"

// assign initial values
in_conference::in_conference (String ck, String tle, int yr, String ed,
                              String pub, String org, String pt, String aut,
                              String pgs)
{
    citekey = ck;
    title = tle;
    year = yr;
    editor = ed;
    publisher = pub;
    organisation = org;
    paper_title = pt;
    author = aut;
    pages = pgs;
}

void in_conference::set_citekey (String ck)
{
    citekey = ck;
}

void in_conference::set_title (String tle)
{
    title = tle;
}

void in_conference::set_year (int yr)
{
    year = yr;
}
void in_conference::set_editor (String ed)
{
    editor = ed;
}
```

```
void in_conference::set_publisher (String pub)
{
     publisher = pub;
}

void in_conference::set_organisation (String org)
{
     organisation = org;
}

void in_conference::set_paper_title (String pt)
{
     paper_title = pt;
}

void in_conference::set_author (String aut)
{
     author = aut;
}

void in_conference::set_pages (String pgs)
{
     pages = pgs;
}

void in_conference::print () const
{
     cout << endl << "cite key: " << citekey << endl
          << "Title: " << title << endl
          << "Year: "  << year << endl
          << "Editor: " << editor << endl
          << "Publisher: " << publisher << endl
          << "Organisation: " << organisation << endl
          << "Paper Title: " << paper_title << endl
          << "Author: " << author << endl
          << "Pages: " << pages << endl;
}

// thesis.h
#include "string.h"     /* Header file for the String Class */

class thesis
{
 private:
     String citekey;
     String title;
     int year;
     String author;
     String institution;

 public:
     /* thesis constructor */
     thesis (String, String, int, String, String);

     void set_citekey (String);
     void set_title (String);
     void set_year (int);
     void set_author (String);
     void set_institution (String);
     void print () const;
};

// thesis.cc
#include <iostream.h>
#include "thesis.h"
```

```
// assign initial values
thesis::thesis (String ck, String tle, int yr, String aut, String inst)
{
     citekey = ck;
     title = tle;
     year = yr;
     author = aut;
     institution = inst;
}

void thesis::set_citekey (String ck)
{
     citekey = ck;
}

void thesis::set_title (String tle)
{
     title = tle;
}

void thesis::set_year (int yr)
{
     year = yr;
}

void thesis::set_author (String aut)
{
     author = aut;
}

void thesis::set_institution (String inst)
{
     institution = inst;
}

void thesis::print () const
{
     cout << endl << "cite key: " << citekey << endl
          << "Title: " << title << endl
          << "Year: "  << year << endl
          << "Author: " << author << endl
          << "Academic Institution: " << institution << endl;
}

// main.cc
Same as inheritance program version
```

D.3 Instructions for second experiment

D.3.1 Instruction Overview

Please read these instructions carefully and in full before moving to the next stage. Any questions you have will be answered by your monitor.

1. Check that you have 2 packets, numbered 1 and 2, and a sheet entitled "Getting Started".

   If you do not have these indicate this to your monitor.

2. When instructed, open the packet numbered 1 which contains information about a program and a modification you are asked to make to it. You have 10 minutes to read this information, and may ask any questions you have during this stage. Please ensure, however, that you understand what is required before moving on.

3. Packet 2 contains the source code you will be modifying. You may approach the modification in any way you see fit - there are no constraints being placed on you. However, please make sure you work as quickly and carefully as you can.

   Important: if you wish to write anything down please do so on the source code listing.

   Once you feel you have completed the modification specified in packet 1, indicate this to a monitor who will examine your program output. If your modification is correct and produces the desired output then you have completed the task, otherwise you should continue making the modification until you do successfully complete the task.

Exceptional Circumstances

Circumstances may arise when you need to stop the practical test for a short period. Such circumstances include a workstation crash. Report any such untoward circumstances to a monitor who will attempt to deal with the problem.

Finishing Off

When you have successfully completed the required task, a monitor will collect your program listing and give you a questionnaire to complete. Please answer the questionnaire as accurately as possible and hand it in before you leave.

D.3.2 General Information (Second experiment)

Program functionality: the program creates objects from the subset of a university population, namely undergraduate and postgraduate students, secretary, lecturer, professor, technician, senior technician and supervisor. It then manipulates some of these objects through their member functions.

Each member of the university community has a set of particular attributes, some of which are common to all members. These communal attributes can be determined from the program listing.

Modification Overview

Program extensibility: you are required to extend the program to deal with a new member, director, of the university community, ie write a class director, and create an instance of this class to show your code performs correctly. See the Detailed Specification section for a complete description of this modification.

File structure

The file structure consists of a header and implementation file for each class, and a main file. These files are conventionally named: that is header files are filename.h, implementation files are filename.cc, and the main file is main.cc.

Furthermore, there exist files in your directory which, while they must present for the compilation to work, **do not need to be modified or even examined**. These files are: string.h and string.cc (they simply provide the means to store a sequence of characters as a string object), and compile (used to compile the program - see next section).

In addition to the string class declarations, the following constants have been declared in string.h:

```
const int this_year = 94
const float relief_tax = 15.0      const float basic_tax = 20.0
const float normal_tax = 25.0      const float upper_tax = 40.0
```

These constants are used throughout the program when calculating employees take home pay.

Compiling and Executing the Program

A shell script has been introduced which saves your files each time you compile your code. This enables us to follow your approach to the modification. Please make sure you compile the program by typing:

```
compile *.cc
```

Note: compile is used in place of CC.

Compilation errors will be reported in the usual way, and the executable will be:

```
a.out
```

Detailed Specification

1. Write a class **director** who has the following attributes:

String last_name;	String first_name;
unsigned age;	String department;
unsigned staff_id_number;	float annual_salary;
String subordinates;	String office;

Your director class should be able to construct an object using parameters, and should be able to set all of its attributes when required, just like any of the other classes. It should also calculate the director's salary after deducting taxes at the **upper tax** band rate. Furthermore, when the director object is displayed to the screen it should display all its private data in a fashion similar to other objects (see the Program Output section for clarification of this output).

You must write your code as a header file called director.h, and an implementation file called director.cc. Once you have written the appropriate code, add the lines

```
director Wilson("Wilson", "Michael", 55, "Science Departments",
                671234, 53995, "All science employees", "");
Wilson.print();
Wilson.set_office("L11.01");
Wilson.print();
```

to function main, in file main.cc. This will create your director object, display it, use the member function which sets the office attribute, and then display the object again to make sure the desired effect has been achieved.

Program Output

You may wish to compile the program before you make any modifications and examine the program output - a secretary and a supervisor object are output.

The actual program output **after** a correct modification should be:

```
Name: Jones, Ann
Age: 21
Department: English
Staff Id. #: 100002
Hourly Wage: $5.55
Hourly wage after tax: $4.7175
Boss: GTB

Name: Johnson, Robert
Age: 47
Department: Chemical Engineering
Staff Id. #: 450000
Annual Salary: $40000
Salary after tax: $30000
Subordinates: Dobbs, Smith & White

Name: Wilson, Michael
Age: 55
Department: Science Departments
Staff Id. #: 671234
Annual Salary: $53995
Salary after tax: $32397
Subordinates : All science employees
Office:

Name: Wilson, Michael
Age: 55
Department: Science Departments
Staff Id. #: 671234
Annual Salary: $53995
Salary after tax: $32397
Subordinates : All science employees
Office: L11.01
```

If you have a problem viewing the program output because it scrolls off the screen, remember you can redirect output to more as follows:

```
a.out | more
```

When you feel you have completed the modification satisfactorily, raise your hand and demonstrate your output to a monitor.

Getting Started

You should run your preferred editor in one window and use a separate window for issuing compilation commands and execution commands. Make sure to `cd C++/Proj2` in both windows.

Important: before you open Packet 2, it is imperative that you script your entire practical session. In the window that you plan to use for compilation and execution commands, type:

```
record
```

This will start a script file called typescript which records your workstation session, and changes your prompt to look something like

```
maxwell-04 jd 1:55pm %
```

If this does not happen when you type `record` please raise your hand and tell a monitor.

Important: At the end of the session, please type

```
exit
```

to finish the auto recording of your programming efforts.

You should now understand what is required of you, and be ready to open Packet 2.

D.4 Code for second experiment

Inheritance program version

```
// lecturer.h
class lecturer : public staff
{
 private:
     float annual_salary;

 public:
     /* lecturer constructor */
     lecturer (const String, const String, unsigned, const String,
               unsigned, float);

     /* default lecturer constructor */
     lecturer ();

     void set_annual_salary (const float);
     void taxable_salary () const;
     void print () const;
};

// lecturer.cc
#include <iostream.h>
#include "univ_community.h"
#include "staff.h"
#include "lecturer.h"

// assign initial values
lecturer::lecturer (const String ln, const String fn, unsigned a,
                    const String dept, unsigned sin, float as) :
                    staff (ln, fn, a, dept, sin)
{
     annual_salary = as;
}

// no initial values, assign defaults
lecturer::lecturer () :
staff ()
{
     annual_salary = 0.00;
}

void lecturer::set_annual_salary (const float as)
{
     annual_salary = as;
}

void lecturer::taxable_salary () const
{
     float tax_rate = 100 - normal_tax;
     float tax_sal;

     tax_sal = tax_rate/100 * annual_salary;
     cout << "Salary after tax: $" << tax_sal << endl;
}

void lecturer::print () const
{
     staff::print();
     cout << "Annual Salary: $" << annual_salary << endl;
     lecturer::taxable_salary();
}
```

```
// pgrad.h
class pgrad : public student
{
 private:
     String degree;
     unsigned duration;

 public:
     /* pgrad constructor */
     pgrad (const String, const String, unsigned, const String,
            unsigned, unsigned, const String, unsigned);

     /* default pgrad constructor */
     pgrad ();

     void set_degree (const String);
     void set_duration (const unsigned);
     void print () const;
};

// pgrad.cc
#include <iostream.h>
#include "univ_community.h"
#include "student.h"
#include "pgrad.h"

// assign initial values
pgrad::pgrad (const String ln, const String fn, unsigned a, const String dept,
              unsigned reg_numb, unsigned yr, const String deg, unsigned dur) :
              student (ln, fn, a, dept, reg_numb, yr)
{
     degree = deg;
     duration = dur;
}

// no initial values, assign defaults
pgrad::pgrad () :
student (), degree ()
{
     duration = 0;
}

void pgrad::set_degree (const String deg)
{
     degree = deg;
}

void pgrad::set_duration (const unsigned dur)
{
     duration = dur;
}

void pgrad::print () const
{
     student::print();
     cout << "Degree: " << degree << endl
          << "Duration (years): " << duration <<endl;
}

// professor.h
class professor : public lecturer
{
 private:
     unsigned research_grant_number;
```

```
 public:
     /* professor constructor */
     professor (const String, const String, unsigned, const String,
                unsigned, float, unsigned);

     /* default staff constructor */
     professor ();

     void set_research_grant (const unsigned);
     void print () const;
};

// professor.cc
#include <iostream.h>
#include "univ_community.h"
#include "staff.h"
#include "lecturer.h"
#include "professor.h"

// assign initial values
professor::professor (const String ln, const String fn, unsigned a,
                      const String dept, unsigned sin, float as, unsigned r) :
                      lecturer (ln, fn, a, dept, sin, as)
{
     research_grant_number = r;
}

// no initial values, assign defaults
professor::professor () :
lecturer ()
{
     research_grant_number = 0;
}

void professor::set_research_grant (const unsigned r)
{
     research_grant_number = r;
}

void professor::print () const
{
     lecturer::print();
     cout << "Research Grant #: " << research_grant_number << endl;
}

// secretary.h
class secretary : public staff
{
 private:
     float hourly_wage;
     String boss;

 public:
     /* secretary constructor */
     secretary (const String, const String, unsigned, const String,
                unsigned, float, const String);

     /* default secretary constructor */
     secretary ();

     void set_hourly_wage (const float);
     void set_boss (const String);
     void taxable_salary () const;
     void print () const;
```

```
};

// secretary.cc
#include <iostream.h>
#include "univ_community.h"
#include "staff.h"
#include "secretary.h"

// assign initial values
secretary::secretary (const String ln, const String fn, unsigned a,
                      const String dept, unsigned sin, float hw,
                      const String bs) :
                      staff (ln, fn, a, dept, sin)
{
     hourly_wage = hw;
     boss = bs;
}

// no initial values, assign defaults
secretary::secretary () :
staff (), boss ()
{
     hourly_wage = 0.00;
}

void secretary::set_hourly_wage (const float hw)
{
     hourly_wage = hw;
}

void secretary::set_boss (const String bs)
{
     boss = bs;
}

void secretary::taxable_salary () const
{
     float tax_rate = 100 - relief_tax;
     float tax_sal;

     tax_sal = tax_rate/100 * hourly_wage;
     cout << "Hourly wage after tax: $" << tax_sal << endl;
}

void secretary::print () const
{
     staff::print();
     cout << "Hourly Wage: $" << hourly_wage << endl;
     secretary::taxable_salary();
     cout << "Boss: "<< boss << endl;
}

// senior_technician.h
class senior_technician : public technician
{
 private:
        /* No new private data */

 public:
     /* technician constructor */
     senior_technician (const String, const String, unsigned,
                        const String, unsigned, float);

     /* default technician constructor */
     senior_technician ();
```

```
     void taxable_salary () const;
     void print () const;
};

// senior_technician.cc
#include <iostream.h>
#include "univ_community.h"
#include "staff.h"
#include "technician.h"
#include "senior_technician.h"

// assign initial values
senior_technician::senior_technician (const String ln, const String fn,
                                      unsigned a, const String dept,
                                      unsigned sin, float as) :
                                      technician (ln, fn, a, dept, sin, as)
{
}

// no initial values, assign defaults
senior_technician::senior_technician () :
technician()
{
}

void senior_technician::taxable_salary () const
{
     float tax_rate = 100 - normal_tax;
     float tax_sal;
     float sal = technician::get_annual_salary();

     tax_sal = tax_rate/100 * sal;
     cout << "Salary after tax: $" << tax_sal << endl;
}

void senior_technician::print () const
{
     technician::print ();
}

// staff.h
class staff : public univ_community
{
 private:
     unsigned staff_id_number;

 public:
     /* staff constructor */
     staff (const String, const String, unsigned, const String, unsigned);

     /* default staff constructor */
     staff ();

     void set_staff_id_number (const unsigned);
     void years_at_univ () const;
     void print () const;

     /* Pure virtual function */
     virtual void taxable_salary () const = 0;
};

// staff.cc
#include <iostream.h>
#include <stdlib.h>
```

```
#include "univ_community.h"
#include "staff.h"

// assign initial values
staff::staff (const String ln, const String fn, unsigned a,
              const String dept, unsigned sin) :
              univ_community (ln, fn, a, dept)
{
     staff_id_number = sin;
}

// no initial values, assign defaults
staff::staff () :
univ_community ()
{
     staff_id_number = 0;
}

void staff::set_staff_id_number(const unsigned sin)
{
     staff_id_number = sin;
}

void staff::years_at_univ () const
{
     unsigned years;

     years = this_year - abs(staff_id_number/10000);
     cout << "Years at university: " << years << endl;
}

void staff::print () const
{
     univ_community::print ();
     cout << "Staff Id. #: " << staff_id_number << endl;
}

// student.h
class student : public univ_community
{
 private:
     unsigned registration_number;
     unsigned year;

 public:
     /* student constructor */
     student (const String, const String, unsigned, const String,
              unsigned, unsigned);

     /* default student constructor */
     student ();

     void set_registration_number (const unsigned);
     void set_year (const unsigned);
     void years_at_univ () const;
     void print () const;
};

// student.cc
#include <iostream.h>
#include "univ_community.h"
#include "student.h"

// assign initial values
student::student (const String ln, const String fn, unsigned a,
```

```
                const String dept, unsigned reg_numb, unsigned yr) :
                univ_community (ln, fn, a, dept)
{
     registration_number = reg_numb;
     year = yr;
}

// no initial values, assign defaults
student::student () :
univ_community ()
{
     registration_number = 0;
     year = 0;
}

void student::set_registration_number (const unsigned reg_numb)
{
     registration_number = reg_numb;
}

void student::set_year (const unsigned yr)
{
     year = yr;
}

void student::years_at_univ () const
{
     cout << "Years at University: " << year << endl;
}

void student::print () const
{
     univ_community::print();
     cout << "Registration #: " << registration_number << endl
          << "Year: " << year << endl;
}

// supervisor.h
class supervisor : public senior_technician
{
 private:
     String subordinates;

 public:
     /* supervisor constructor */
     supervisor (const String, const String, unsigned, const String,
                 unsigned, float, const String);

     /* default technician constructor */
     supervisor ();

     void set_subordinates (const String);
     void print () const;
};

// supervisor.cc
#include <iostream.h>
#include "univ_community.h"
#include "staff.h"
#include "technician.h"
#include "senior_technician.h"
#include "supervisor.h"

// assign initial values
supervisor::supervisor (const String ln, const String fn, unsigned a,
```

```
                        const String dept, unsigned sin, float as,
                        const String subs) :
                        senior_technician (ln, fn, a, dept, sin, as)
{
     subordinates = subs;
}

// no initial values, assign defaults
supervisor::supervisor () :
senior_technician(), subordinates()
{
}

void supervisor::set_subordinates (const String subs)
{
     subordinates = subs;
}

void supervisor::print () const
{
     senior_technician::print ();
     cout << "Subordinates : " << subordinates << endl;
}

// technician.h
class technician : public staff
{
 private:
     float annual_salary;

 public:
     /* technician constructor */
     technician (const String, const String, unsigned, const String,
                 unsigned, float);

     /* default technician constructor */
     technician ();

     void set_annual_salary (const float);
     float get_annual_salary () const;
     void taxable_salary () const;
     void print () const;
};

// technician.cc
#include <iostream.h>
#include "univ_community.h"
#include "staff.h"
#include "technician.h"

// assign initial values
technician::technician (const String ln, const String fn, unsigned a,
                        const String dept, unsigned sin, float as) :
                        staff (ln, fn, a, dept, sin)
{
     annual_salary = as;
}

// no initial values, assign defaults
technician::technician () :
staff ()
{
     annual_salary = 0.00;
}
```

```
void technician::set_annual_salary (const float as)
{
     annual_salary = as;
}

float technician::get_annual_salary() const
{
     return annual_salary;
}

void technician::taxable_salary () const
{
     float tax_rate = 100 - basic_tax;
     float tax_sal;

     tax_sal = tax_rate/100 * annual_salary;
     cout << "Salary after tax: $" << tax_sal << endl;
}

void technician::print () const
{
     staff::print ();
     cout << "Annual Salary: $" << annual_salary << endl;

     // use this pointer in order to call the correct taxable_salary function
     // when dealing with derived classes
     this -> taxable_salary ();
}

// ugrad.h
class ugrad : public student
{
 private:
     String course;

 public:
     /* ugrad constructor */
     ugrad (const String, const String, unsigned, const String,
            unsigned, unsigned, const String);

     /* default ugrad constructor */
     ugrad ();

     void set_course (const String);
     void print () const;
};

// ugrad.cc
#include <iostream.h>
#include "univ_community.h"
#include "student.h"
#include "ugrad.h"

// assign initial values
ugrad::ugrad (const String ln, const String fn, unsigned a,
              const String dept, unsigned reg_numb, unsigned yr,
              const String cs) :
              student (ln, fn, a, dept, reg_numb, yr)
{
     course = cs;
}

// no initial values, assign defaults
ugrad::ugrad () :
student (), course()
```

```
{
}

void ugrad::set_course (const String cs)
{
     course = cs;
}

void ugrad::print () const
{
     student::print();
     cout << "University course: " << course << endl;
}

// univ_community.h
#include "string.h"    /* Header file for the String Class */

class univ_community
{
 private:
     String last_name; /* String object which stores last name */
     String first_name;
     unsigned age;
     String department;

 public:
     /* univ_community constructor */
     univ_community (const String, const String, unsigned, const String);

     /* default univ_community constructor */
     univ_community ();

     void set_last_name (const String);
     void set_first_name (const String);
     void set_age (const unsigned);
     void set_department (const String);

     /* allow print to be overwritten by derived classes: make it virtual */
     virtual void print () const;
     /* pure virtual function */
     virtual void years_at_univ () const = 0;
};

// univ_community.cc
#include <iostream.h>
#include "univ_community.h"

// assign initial values
univ_community::univ_community (const String ln, const String fn,
                                unsigned a, const String dept)
{
     last_name = ln;
     first_name = fn;
     age = a;
     department = dept;
}

/* no initial values, assign defaults */
univ_community::univ_community () :
last_name(), first_name(), department()
{
     age = 0;
}

void univ_community::set_last_name (const String last_n)
```

```
{
     last_name = last_n;
}

void univ_community::set_first_name (const String first_n)
{
     first_name = first_n;
}

void univ_community::set_age (const unsigned a)
{
     age = a;
}

void univ_community::set_department(const String dept)
{
     department = dept;
}

void univ_community::print () const
{
     cout << endl << "Name: " << last_name << ", "
          << first_name << endl
          << "Age: " << age << endl
          << "Department: " << department << endl;
}

// main.cc
#include "univ_community.h"     // Header file for Univ_community Class
#include "student.h"            // Header file for Student Class
#include "ugrad.h"              // Header file for Ugrad Class
#include "pgrad.h"              // Header file for Pgrad Class
#include "staff.h"              // Header file for Staff Class
#include "technician.h"         // Header file for Technician Class
#include "senior_technician.h"  // Header file for Senior_technician Class
#include "supervisor.h"         // Header file for Supervisor class
#include "lecturer.h"           // Header file for Lecturer Class
#include "professor.h"          // Header file for Professor Class
#include "secretary.h"          // Header file for Secretary Class

main()
{
     /* create objects and assign values using the constructor */
     pgrad Douglas("Douglas", "Craig", 23, "Marketing", 9263520, 2, "PhD", 3);
     ugrad Watson("Watson", "Andy", 26, "Computer Science", 9164789,
                  4, "Computer and Electronic Systems");
     lecturer Smith("Smith", "David", 34, "Mathematics", 890001, 24995);
     professor Davidson("Davidson", "Joe", 53, "Mathematics", 771234,
                        31995, 12);
     technician Dobbs("Dobbs", "Davie", 27, "Chemical Engineering",
                      740021, 13450);
     senior_technician White("White", "Jimmy", 43, "Chemical Engineering",
                             893212, 19950);
     supervisor Johnson("Johnson", "Robert", 47, "Chemical Engineering",
                        450000, 40000, "Dobbs, Smith & White");

     /* create the object using the default constructor */
     secretary Jones;
     /* set the variables using the object's member functions */
     Jones.set_first_name("Ann");
     Jones.set_last_name("Jones");
     Jones.set_age(21);
     Jones.set_department("English");
     Jones.set_staff_id_number(100002);
     Jones.set_boss("GTB");
```

```
     Jones.set_hourly_wage(5.55);

     /* print the Jones object after all the values have been set */
     Jones.print();
     /* print the Johnson object */
     Johnson.print();
}
```

Flat program version

```
// lecturer.h
#include "string.h"    /* Header file for the String Class */

class lecturer
{
 private:
     String last_name; /* String object which stores last name */
     String first_name;
     unsigned age;
     String department;
     unsigned staff_id_number;
     float annual_salary;

 public:
     /* lecturer constructor */
     lecturer (const String, const String, unsigned, const String,
               unsigned, float);

     /* default lecturer constructor */
     lecturer ();

     void set_last_name (const String);
     void set_first_name (const String);
     void set_age (const unsigned);
     void set_department (const String);
     void set_staff_id_number (const unsigned);
     void set_annual_salary (const float);
     void years_at_univ () const;
     void taxable_salary () const;
     void print () const;
};

// lecturer.cc
#include <iostream.h>
#include <stdlib.h>
#include "lecturer.h"

// assign initial values
lecturer::lecturer (const String ln, const String fn, unsigned a,
                    const String dept, unsigned sin, float as)
{
     last_name = ln;
     first_name = fn;
     age = a;
     department = dept;
     staff_id_number = sin;
     annual_salary = as;
}

// no initial values, assign defaults
lecturer::lecturer () :
last_name(), first_name(), department()
{
     age = 0;
     staff_id_number = 0;
```

```
     annual_salary = 0.00;
}

void lecturer::set_last_name (const String last_n)
{
     last_name = last_n;
}

void lecturer::set_first_name (const String first_n)
{
     first_name = first_n;
}

void lecturer::set_age (const unsigned a)
{
     age = a;
}

void lecturer::set_department(const String dept)
{
     department = dept;
}

void lecturer::set_staff_id_number(const unsigned sin)
{
     staff_id_number = sin;
}

void lecturer::set_annual_salary (const float as)
{
     annual_salary = as;
}

void lecturer::years_at_univ () const
{
     unsigned years;

     years = this_year - abs(staff_id_number/10000);
     cout << "Years at university: " << years << endl;
}

void lecturer::taxable_salary () const
{
     float tax_rate = 100 - normal_tax;
     float tax_sal;

     tax_sal = tax_rate/100 * annual_salary;
     cout << "Salary after tax: $" << tax_sal << endl;
}

void lecturer::print () const
{
     cout << endl << "Name: " << last_name << ", "
          << first_name << endl
          << "Age: " << age << endl
          << "Department: " << department << endl
          << "Staff Id. #: " << staff_id_number << endl
          << "Annual Salary: $" << annual_salary << endl;
     lecturer::taxable_salary();
}

// pgrad.h
#include "string.h"

class pgrad
```

```
{
 private:
     String last_name; /* String object which stores last name */
     String first_name;
     unsigned age;
     String department;
     unsigned registration_number;
     unsigned year;
     String degree;
     unsigned duration;

 public:
     /* pgrad constructor */
     pgrad (const String, const String, unsigned, const String,
            unsigned, unsigned, const String, unsigned);

     /* default pgrad constructor */
     pgrad ();

     void set_last_name (const String);
     void set_first_name (const String);
     void set_age (const unsigned);
     void set_department (const String);
     void set_registration_number (const unsigned);
     void set_year (const unsigned);
     void set_degree (const String);
     void set_duration (const unsigned);
     void years_at_univ () const;
     void print () const;
};

// pgrad.cc
#include <iostream.h>
#include "pgrad.h"

// assign initial values
pgrad::pgrad (const String ln, const String fn, unsigned a, const String dept,
              unsigned reg_numb, unsigned yr, const String deg, unsigned dur)
{
     last_name = ln;
     first_name = fn;
     age = a;
     department = dept;
     registration_number = reg_numb;
     year = yr;
     degree = deg;
     duration = dur;
}

// no initial values, assign defaults
pgrad::pgrad () :
last_name(), first_name(), department(), degree()
{
     age = 0;
     registration_number = 0;
     year = 0;
     duration = 0;
}

void pgrad::set_last_name (const String last_n)
{
     last_name = last_n;
}

void pgrad::set_first_name (const String first_n)
```

```
{
     first_name = first_n;
}

void pgrad::set_age (const unsigned a)
{
     age = a;
}

void pgrad::set_department(const String dept)
{
     department = dept;
}

void pgrad::set_registration_number (const unsigned reg_numb)
{
     registration_number = reg_numb;
}

void pgrad::set_year (const unsigned yr)
{
     year = yr;
}

void pgrad::set_degree (const String deg)
{
     degree = deg;
}

void pgrad::set_duration (const unsigned dur)
{
     duration = dur;
}

void pgrad::years_at_univ () const
{
     cout << "Years at University: " << year << endl;
}

void pgrad::print () const
{
     cout << endl << "Name: " << last_name << ", "
          << first_name << endl
          << "Age: " << age << endl
          << "Department: " << department << endl
          << "Registration #: " << registration_number << endl
          << "Year: " << year << endl
          << "Degree: " << degree << endl
          << "Duration (years): " << duration << endl;
}

// professor.h
#include "string.h"    /* Header file for the String Class */

class professor
{
 private:
     String last_name; /* String object which stores last name */
     String first_name;
     unsigned age;
     String department;
     unsigned staff_id_number;
     float annual_salary;
     unsigned research_grant_number;
```

```
 public:
     /* professor constructor */
     professor (const String, const String, unsigned, const String,
               unsigned, float, unsigned);

     /* default professor constructor */
     professor ();

     void set_last_name (const String);
     void set_first_name (const String);
     void set_age (const unsigned);
     void set_department (const String);
     void set_staff_id_number (const unsigned);
     void set_annual_salary (const float);
     void set_research_grant (const unsigned);
     void years_at_univ () const;
     void taxable_salary () const;
     void print () const;
};

// professor.cc
#include <iostream.h>
#include <stdlib.h>
#include "professor.h"

// assign initial values
professor::professor (const String ln, const String fn, unsigned a,
                      const String dept, unsigned sin, float as, unsigned r)
{
     last_name = ln;
     first_name = fn;
     age = a;
     department = dept;
     staff_id_number = sin;
     annual_salary = as;
     research_grant_number = r;
}

// no initial values, assign defaults
professor::professor () :
last_name(), first_name(), department()
{
     age = 0;
     staff_id_number = 0;
     annual_salary = 0.00;
     research_grant_number = 0;
}

void professor::set_last_name (const String last_n)
{
     last_name = last_n;
}

void professor::set_first_name (const String first_n)
{
     first_name = first_n;
}

void professor::set_age (const unsigned a)
{
     age = a;
}

void professor::set_department(const String dept)
{
```

```
     department = dept;
}

void professor::set_staff_id_number(const unsigned sin)
{
     staff_id_number = sin;
}

void professor::set_annual_salary (const float as)
{
     annual_salary = as;
}

void professor::set_research_grant (const unsigned r)
{
     research_grant_number = r;
}

void professor::years_at_univ () const
{
     unsigned years;

     years = this_year - abs(staff_id_number/10000);
     cout << "Years at university: " << years << endl;
}

void professor::taxable_salary () const
{
     float tax_rate = 100 - normal_tax;
     float tax_sal;

     tax_sal = tax_rate/100 * annual_salary;
     cout << "Salary after tax: $" << tax_sal << endl;
}

void professor::print () const
{
     cout << endl << "Name: " << last_name << ", "
          << first_name << endl
          << "Age: " << age << endl
          << "Department: " << department << endl
          << "Staff Id. #: " << staff_id_number << endl
          << "Annual Salary: $" << annual_salary << endl;
     professor::taxable_salary();
     cout << "Research Grant #: " << research_grant_number << endl;
}

// secretary.h
#include "string.h"    /* Header file for the String Class */

class secretary
{
 private:
     String last_name; /* String object which stores last name */
     String first_name;
     unsigned age;
     String department;
     unsigned staff_id_number;
     float hourly_wage;
     String boss;

 public:
     /* secretary constructor */
     secretary (const String, const String, unsigned, const String,
                unsigned, float, const String);
```

```
     /* default secretary constructor */
     secretary ();

     void set_last_name (const String);
     void set_first_name (const String);
     void set_age (const unsigned);
     void set_department (const String);
     void set_staff_id_number (const unsigned);
     void set_hourly_wage (const float);
     void set_boss (const String);
     void years_at_univ () const;
     void taxable_salary () const;
     void print () const;
};

// secretary.cc
#include <iostream.h>
#include <stdlib.h>
#include "secretary.h"

// assign initial values
secretary::secretary (const String ln, const String fn, unsigned a,
                      const String dept, unsigned sin, float hw,
                      const String bs)
{
     last_name = ln;
     first_name = fn;
     age = a;
     department = dept;
     staff_id_number = sin;
     hourly_wage = hw;
     boss = bs;
}

// no initial values, assign defaults
secretary::secretary () :
last_name(), first_name(), department(), boss()
{
     age = 0;
     staff_id_number = 0;
     hourly_wage = 0.00;
}

void secretary::set_last_name (const String last_n)
{
     last_name = last_n;
}

void secretary::set_first_name (const String first_n)
{
     first_name = first_n;
}

void secretary::set_age (const unsigned a)
{
     age = a;
}

void secretary::set_department(const String dept)
{
     department = dept;
}

void secretary::set_staff_id_number(const unsigned sin)
```

```
{
     staff_id_number = sin;
}
void secretary::set_hourly_wage (const float hw)
{
     hourly_wage = hw;
}

void secretary::set_boss (const String bs)
{
     boss = bs;
}

void secretary::years_at_univ () const
{
     unsigned years;

     years = this_year - abs(staff_id_number/10000);
     cout << "Years at university: " << years << endl;
}

void secretary::taxable_salary () const
{
     float tax_rate = 100 - relief_tax;
     float tax_sal;

     tax_sal = tax_rate/100 * hourly_wage;
     cout << "Hourly wage after tax: $" << tax_sal << endl;
}

void secretary::print () const
{
     cout << endl << "Name: " << last_name << ", "
          << first_name << endl
          << "Age: " << age << endl
          << "Department: " << department << endl
          << "Staff Id. #: " << staff_id_number << endl
          << "Hourly Wage: $" << hourly_wage << endl;
     secretary::taxable_salary();
     cout << "Boss: "<< boss << endl;
}

// senior_technician.h
#include "string.h"    /* Header file for the String Class */

class senior_technician
{
 private:
     String last_name; /* String object which stores last name */
     String first_name;
     unsigned age;
     String department;
     unsigned staff_id_number;
     float annual_salary;

 public:
     /* senior_technician constructor */
     senior_technician (const String, const String, unsigned, const String,
                        unsigned, float);

     /* default senior_technician constructor */
     senior_technician ();

     void set_last_name (const String);
     void set_first_name (const String);
```

```
     void set_age (const unsigned);
     void set_department (const String);
     void set_staff_id_number (const unsigned);
     void set_annual_salary (const float);
     float get_annual_salary () const;
     void years_at_univ () const;
     void taxable_salary () const;
     void print () const;
};

// senior_technician.cc
#include <iostream.h>
#include <stdlib.h>
#include "senior_technician.h"

// assign initial values
senior_technician::senior_technician (const String ln, const String fn,
                                      unsigned a, const String dept,
                                      unsigned sin, float as)
{
     last_name = ln;
     first_name = fn;
     age = a;
     department = dept;
     staff_id_number = sin;
     annual_salary = as;
}

// no initial values, assign defaults
senior_technician::senior_technician () :
last_name(), first_name(), department()
{
     age = 0;
     staff_id_number = 0;
     annual_salary = 0.00;
}

void senior_technician::set_last_name (const String last_n)
{
     last_name = last_n;
}

void senior_technician::set_first_name (const String first_n)
{
     first_name = first_n;
}

void senior_technician::set_age (const unsigned a)
{
     age = a;
}

void senior_technician::set_department(const String dept)
{
     department = dept;
}

void senior_technician::set_staff_id_number(const unsigned sin)
{
     staff_id_number = sin;
}

void senior_technician::set_annual_salary (const float as)
{
     annual_salary = as;
```

```
}

float senior_technician::get_annual_salary() const
{
     return annual_salary;
}

void senior_technician::years_at_univ () const
{
     unsigned years;

     years = this_year - abs(staff_id_number/10000);
     cout << "Years at university: " << years << endl;
}

void senior_technician::taxable_salary () const
{
     float tax_rate = 100 - normal_tax;
     float tax_sal;

     tax_sal = tax_rate/100 * annual_salary;
     cout << "Salary after tax: $" << tax_sal << endl;
}

void senior_technician::print () const
{
     cout << endl << "Name: " << last_name << ", "
          << first_name << endl
          << "Age: " << age << endl
          << "Department: " << department << endl
          << "Staff Id. #: " << staff_id_number << endl
          << "Annual Salary: $" << annual_salary << endl;

     // use this pointer in order to call the correct taxable_salary function
     // when dealing with derived classes
     this->taxable_salary();
}

// supervisor.h
#include "string.h"    /* Header file for the String Class */

class supervisor
{
 private:
     String last_name; /* String object which stores last name */
     String first_name;
     unsigned age;
     String department;
     unsigned staff_id_number;
     float annual_salary;
     String subordinates;

 public:
     /* supervisor constructor */
     supervisor (const String, const String, unsigned, const String,
                 unsigned, float, const String);

     /* default supervisor constructor */
     supervisor ();

     void set_last_name (const String);
     void set_first_name (const String);
     void set_age (const unsigned);
     void set_department (const String);
     void set_staff_id_number (const unsigned);
```

```
     void set_annual_salary (const float);
     float get_annual_salary () const;
     void set_subordinates (const String);
     void years_at_univ () const;
     void taxable_salary () const;
     void print () const;
};

// supervisor.cc
#include <iostream.h>
#include <stdlib.h>
#include "supervisor.h"

// assign initial values
supervisor::supervisor (const String ln, const String fn, unsigned a,
                        const String dept, unsigned sin, float as,
                        const String subs)
{
     last_name = ln;
     first_name = fn;
     age = a;
     department = dept;
     staff_id_number = sin;
     annual_salary = as;
     subordinates = subs;
}

// no initial values, assign defaults
supervisor::supervisor () :
last_name(), first_name(), department(), subordinates()
{
     age = 0;
     staff_id_number = 0;
     annual_salary = 0.00;
}

void supervisor::set_last_name (const String last_n)
{
     last_name = last_n;
}

void supervisor::set_first_name (const String first_n)
{
     first_name = first_n;
}

void supervisor::set_age (const unsigned a)
{
     age = a;
}

void supervisor::set_department(const String dept)
{
     department = dept;
}

void supervisor::set_staff_id_number(const unsigned sin)
{
     staff_id_number = sin;
}

void supervisor::set_annual_salary (const float as)
{
     annual_salary = as;
}
```

```
float supervisor::get_annual_salary() const
{
     return annual_salary;
}

void supervisor::set_subordinates (const String subs)
{
     subordinates = subs;
}

void supervisor::years_at_univ () const
{
     unsigned years;

     years = this_year - abs(staff_id_number/10000);
     cout << "Years at university: " << years << endl;
}

void supervisor::taxable_salary () const
{
     float tax_rate = 100 - normal_tax;
     float tax_sal;

     tax_sal = tax_rate/100 * annual_salary;
     cout << "Salary after tax: $" << tax_sal << endl;
}

void supervisor::print () const
{
     cout << endl << "Name: " << last_name << ", "
          << first_name << endl
          << "Age: " << age << endl
          << "Department: " << department << endl
          << "Staff Id. #: " << staff_id_number << endl
          << "Annual Salary: $" << annual_salary << endl;

     // use this pointer in order to call the correct taxable_salary function
     // when dealing with derived classes
     this->taxable_salary();
     cout << "Subordinates: " << subordinates << endl;
}

// technician.h
#include "string.h"    /* Header file for the String Class */

class technician
{
 private:
     String last_name; /* String object which stores last name */
     String first_name;
     unsigned age;
     String department;
     unsigned staff_id_number;
     float annual_salary;

 public:
     /* technician constructor */
     technician (const String, const String, unsigned, const String,
                 unsigned, float);

     /* default technician constructor */
     technician ();

     void set_last_name (const String);
```

```
     void set_first_name (const String);
     void set_age (const unsigned);
     void set_department (const String);
     void set_staff_id_number (const unsigned);
     void set_annual_salary (const float);
     float get_annual_salary () const;
     void years_at_univ () const;
     void taxable_salary () const;
     void print () const;
};

// technician.cc
#include <iostream.h>
#include <stdlib.h>
#include "technician.h"

// assign initial values
technician::technician (const String ln, const String fn, unsigned a,
                        const String dept, unsigned sin, float as)
{
     last_name = ln;
     first_name = fn;
     age = a;
     department = dept;
     staff_id_number = sin;
     annual_salary = as;
}

// no initial values, assign defaults
technician::technician () :
last_name(), first_name(), department()
{
     age = 0;
     staff_id_number = 0;
     annual_salary = 0.00;
}

void technician::set_last_name (const String last_n)
{
     last_name = last_n;
}

void technician::set_first_name (const String first_n)
{
     first_name = first_n;
}

void technician::set_age (const unsigned a)
{
     age = a;
}

void technician::set_department(const String dept)
{
     department = dept;
}

void technician::set_staff_id_number(const unsigned sin)
{
     staff_id_number = sin;
}

void technician::set_annual_salary (const float as)
{
     annual_salary = as;
```

```
}

float technician::get_annual_salary() const
{
     return annual_salary;
}

void technician::years_at_univ () const
{
     unsigned years;

     years = this_year - abs(staff_id_number/10000);
     cout << "Years at university: " << years << endl;
}

void technician::taxable_salary () const
{
     float tax_rate = 100 - basic_tax;
     float tax_sal;

     tax_sal = tax_rate/100 * annual_salary;
     cout << "Salary after tax: $" << tax_sal << endl;
}

void technician::print () const
{
     cout << endl << "Name: " << last_name << ", "
          << first_name << endl
          << "Age: " << age << endl
          << "Department: " << department << endl
          << "Staff Id. #: " << staff_id_number << endl
          << "Annual Salary: $" << annual_salary << endl;
     // use this pointer in order to call the correct taxable_salary function
     // when dealing with derived classes
     this->taxable_salary();
}

// ugrad.h
#include "string.h"

class ugrad
{
 private:
     String last_name; /* String object which stores last name */
     String first_name;
     unsigned age;
     String department;
     unsigned registration_number;
     unsigned year;
     String course;

 public:
     /* ugrad constructor */
     ugrad (const String, const String, unsigned, const String,
            unsigned, unsigned, const String);

     /* default ugrad constructor */
     ugrad ();

     void set_last_name (const String);
     void set_first_name (const String);
     void set_age (const unsigned);
     void set_department (const String);
     void set_registration_number (const unsigned);
     void set_year (const unsigned);
```

```
     void set_course (const String);
     void years_at_univ () const;
     void print () const;
};

// ugrad.cc
#include <iostream.h>
#include "ugrad.h"

// assign initial values
ugrad::ugrad (const String ln, const String fn, unsigned a,
              const String dept, unsigned reg_numb, unsigned yr,
              const String cs)
{
     last_name = ln;
     first_name = fn;
     age = a;
     department = dept;
     registration_number = reg_numb;
     year = yr;
     course = cs;
}

// no initial values, assign defaults
ugrad::ugrad () :
last_name(), first_name(), department(), course()
{
     age = 0;
     registration_number = 0;
     year = 0;
}

void ugrad::set_last_name (const String last_n)
{
     last_name = last_n;
}

void ugrad::set_first_name (const String first_n)
{
     first_name = first_n;
}

void ugrad::set_age (const unsigned a)
{
     age = a;
}

void ugrad::set_department(const String dept)
{
     department = dept;
}

void ugrad::set_registration_number (const unsigned reg_numb)
{
     registration_number = reg_numb;
}

void ugrad::set_year (const unsigned yr)
{
     year = yr;
}

void ugrad::set_course (const String cs)
{
     course = cs;
```

```
}

void ugrad::years_at_univ () const
{
     cout << "Years at University: " << year << endl;
}

void ugrad::print () const
{
     cout << endl << "Name: " << last_name << ", "
          << first_name << endl
          << "Age: " << age << endl
          << "Department: " << department << endl
          << "Registration #: " << registration_number << endl
          << "Year: " << year << endl
          << "University course: " << course << endl;
}
```

D.5 Debriefing questionnaire

Please answer this questionnaire as honestly as you can. Anything you write down will be treated confidentially.

Personal Details

Name:

Qualifications: Programming experience:

1. How long into the test did it take you to grasp what was required, eg after reading the instructions, after examining the code, etc?

2. How much trouble, if any, did you have with the C++ syntax?

3. On a scale of 1 to 10 how difficult would you say the modification was (1 very easy, 10 very difficult)?

4. What caused you the most difficulty?

5. Overall, what action would you say took you the most time to perform, for example understanding the code, removing syntax errors, editing the changes, etc?

6. What approach did you adopt to tackle the modification?
   (a) Understanding the code first, then tackling the task.
   (b) Tackle task immediately, and attempt to understand the code as required.
   (c) Cutting and pasting the existing files to meet the required specification.
   (d) Other, please specify:

7. Did you use inheritance or not? Explain why.

8. If you answered **yes** to 7, which class did you use as the parent for the class director? Why did you use this class, and how long did it take to make this decision?

9. How well do you understand the code?

10. What parts of the code, if any, did you not understand?

11. How would you judge the quality of the code you produced compared to the code you were given?

12. Having performed the modification, would you do anything different next time around?

13. Have you learned anything from this? If so what?

14. Any other comments?

Our thanks for your participation in the two experiments. We hope you enjoyed the course and it's of some use to you in the future!

John Daly (PhD student) and the other members of EFoCS.

D.6 Statistical power calculations

Before designing the empirical study it was decided that any effect to be investigated would have to be either a medium or a large effect (at this early level of empirical enquiry, small effects were deemed to be of lesser importance). The multi-method programme of research conducted lead to the belief that the effect size of depth of inheritance fell between these two categories. Statistical power analysis was performed *before* conducting the study using the anticipated subject numbers, a preset α level, and the effect size index introduced by [Cohen, 1969] for medium and large effects (although it was expected that the effect size would fall between these two extremes from the data collected from phases I and II, there was no method to estimate it more accurately because this series of experiments is the first to investigate this phenomenon).

Keppel *et al.* state that the minimum statistical power value adopted should be 0.7 and argue that any value below this represents poor science [Keppel *et al.*, 1992]. It is argued that this is a purist's view, more applicable to areas of science that have been empirically researched for some time, and is not a pragmatic approach for software engineering because: (i) much of the empirical research conducted is the first to investigate any given phenomenon, (ii) it is difficult to accurately estimate the effect size (something which greatly affects the power level), and (iii) the number of programmers required as subjects cannot easily be met. These reasons do not detract from the importance of conducting a statistical power analysis before planning an empirical study: this allows the experimenter to realise their chance of detecting an effect if one exists, provides researchers with insight into the value of the empirical study, and also enables researcher to plan better any external replications. Moreover, it is of extreme importance that if the null hypothesis is not rejected then the researcher considers the statistical power of the experiment: if it is too low then there is no justification to claim that the null hypothesis is in fact true because the probability of Type II error is too great.

D.6.1 First experiment

The attributes used for these calculations were (i) $N = 31$, (ii) $\alpha = 0.05$ (preset alpha level for rejecting H_0 with a two-tailed statistical test), and (iii) $\gamma = 0.5$ & $\gamma = 0.8$ (the medium and large effect size indexes). The power calculations were:

Medium effect: $\delta = 0.5 \times \sqrt{\frac{31}{2}} = 1.97$. Hence, from the appropriate table the derived power level is 0.52.

Large effect: $\delta = 0.8 \times \sqrt{\frac{31}{2}} = 3.15$. Hence, from the appropriate table the derived power level is 0.86.

Hence, it is concluded that there would be approximately a 1 in 2 chance of detecting a medium effect and approximately a 4 in 5 chance of detecting a large effect.

D.6.2 Internal replication

The internal replication was conducted before the results of the first experiment were known. Hence, the estimate of the effect size was the same as that of the first experiment. Other attributes used were: (i) $N = 14.5$, (ii) $\alpha = 0.05$ (preset alpha level for rejecting H_0 with a one-tailed statistical test), and (iii) $\gamma = 0.5$ & $\gamma = 0.8$.

Medium effect: $\delta = 0.5 \times \sqrt{\frac{14.5}{2}} = 1.35$. Hence, from the appropriate table the derived power level is 0.39.

Large effect: $\delta = 0.8 \times \sqrt{\frac{14.5}{2}} = 2.15$. Hence, from the appropriate table the derived power level is 0.70.

It is therefore concluded that there would be about a 40% chance of detecting a medium effect and a 70% chance of detecting a large effect.

D.6.3 Second experiment

The attributes used for these calculations were (i) $N = 15.5$, (ii) $\alpha = 0.05$ (preset alpha level for rejecting H_0 with a one-tailed statistical test), and (iii) $\gamma = 0.5$ & $\gamma = 0.8$ (the medium and large effect size indexes). The power calculations were:

Medium effect: $\delta = 0.5 \times \sqrt{\frac{15.5}{2}} = 1.39$. Hence, from the appropriate table the derived power level is 0.40.

Large effect: $\delta = 0.8 \times \sqrt{\frac{15.5}{2}} = 2.23$. Hence, from the appropriate table the derived power level is 0.72.

These figures are almost the same as the internal replication with approximately a 40% chance of detecting a medium effect and approximately a 70% chance of detecting a large effect.

D.6.4 Conclusions

It is important to note that the above power calculations are only rough estimates; indeed, the fact that non-parametric statistical tests were used reduces the estimated power levels. While power is an important issue in experimentation, especially when the null hypothesis is not rejected, its importance should not be overstated when there is difficulty accurately estimating the effect size, e.g., when conducting primary empirical studies.

For this series of experiments the design is considered to have been carefully planned, the number of subjects relatively high (as far as software engineering experiments are concerned), and the power levels reasonable (given that this is the first empirical study investigating this phenomenon). The fact that the null hypothesis was not rejected for the experiment using a deeper hierarchy is probably due to the fact that the statistical power was not high enough. It does appear that there is an existing effect, evidence of which is provided by the change of direction between the flat and inheritance mean times across the two experiments and replication.

D.7 Inductive analysis databases

This section presents the database for each experimental run used during the inductive analysis, i.e., two databases for the first experiment, one for the internal replication, and one for the second experiment. Explanations of logical variables from the debriefing questionnaires are also presented. Subjects' answers to questionnaires in the first experiment were some what different to answers given in the replication and second experiment: combining the data collected for logical variables were slightly different between experimental runs. These data groupings are explained in full in the sections below.

D.7.1 The first experiment

The logical groupings of data for variables in Tables D.1 and D.2 are now explained. Variable (7) `Experience` (programming experience) is graded as: $1 \equiv$ limited to the MSc course, $2 \equiv$ MSc course and some programming experience outwith it, and $3 \equiv$ a sizable amount of programming experience outwith the MSc course. Variable (9) `InstTime` (time to understand the experimental instructions) is graded as: $1 \equiv$ immediately to < 5 minutes, $2 \equiv 5$ minutes to ≤ 10 minutes, and $3 \equiv > 10$ minutes. Variable (10) `SynDiff` (difficulty with syntax) is graded as: $1 \equiv$ none to very little, 2

	Induction Variables																		
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	A	18	0	y	1	2	y	1	2	2	3	1	a	3	1	2	3	1	1
2	A	39	0	y	5	1	n	-	-	-	-	-	-	-	-	-	-	-	-
3	A	39	0	y	5	2	y	1	2	3	3	1	a	3	2	3	1	2	4
4	A	-	0	y	16	1	n	-	-	-	-	-	-	-	-	-	-	-	-
5	B	-	0	y	14	2	y	3	3	5	3	1	a	1	1	1	1	4	1
6	A	45	0	y	8	1	y	2	2	6	2	1	a	2	3	2	1	3	1
7	A	36	0	y	4	1	y	2	2	6	1	2	a	2	1	2	3	2	1
8	B	98	0	n	18	1	y	3	3	9	2	3	b	1	3	2	2	4	1
9	A	44	0	y	5	3	y	1	2	3	1	3	d	3	1	3	4	3	3
10	A	48	0	y	3	1	y	1	2	3	2	2	a	2	3	3	1	2	1
11	B	-	0	n	16	1	y	3	3	5	2	1	a	1	3	1	1	4	4
12	A	38	0	y	6	2	y	2	1	8	2	1	a	3	1	3	1	1	2
13	B	67	0	y	16	3	y	2	2	5	2	3	a	2	1	2	4	4	1
14	B	38	0	n	4	3	y	1	2	2	4	3	c	3	1	3	2	2	1
15	B	26	0	n	3	1	y	1	1	1	4	1	c	3	1	3	2	2	2
16	A	36	0	y	4	1	y	1	2	3	3	2	a	3	1	3	1	1	2
17	B	-	0	y	12	3	y	2	2	4	2	3	b	2	1	1	4	4	1
18	A	-	0	y	-	1	n	-	-	-	-	-	-	-	-	-	-	-	-
19	B	78	0	n	9	1	y	3	2	2	3	1	a	3	1	3	2	2	4
20	A	57	0	y	7	1	y	1	1	5	2	1	d	2	1	3	1	3	1
21	A	28	0	y	3	1	y	1	1	2	4	1	c	3	1	3	1	1	1
22	B	38	0	n	5	1	y	2	2	2	4	1	a	3	1	3	1	1	2
23	B	29	0	n	2	1	y	2	1	4	4	1	c	2	1	3	5	4	1
24	B	-	0	n	27	1	y	2	1	4	2	2	a	2	1	2	1	4	1
25	A	18	0	y	1	2	y	1	1	1	4	1	d	3	1	3	1	3	1
26	A	92	0	y	17	1	y	1	2	3	2	3	a	3	1	2	1	4	3
27	B	46	0	n	9	1	y	2	1	5	2	1	a	3	1	2	2	1	1
28	B	64	0	y	13	1	y	1	2	4	2	2	a	2	1	3	1	4	1
29	B	-	0	n	17	1	n	-	-	-	-	-	-	-	-	-	-	-	-
30	B	47	0	n	6	1	y	3	1	2	2	2	a	2	1	3	1	1	2
31	A	79	0	y	12	1	y	3	2	7	3	1	a	2	3	3	1	1	1

Table D.1: Raw data for the first run of the first experiment

≡ little to some, and 3 ≡ quite a bit to substantial. Variable (12) `Diff` (what caused the most difficulty) is graded as: 1 ≡ inheritance related difficulties, 2 ≡ syntax and compiler errors, 3 ≡ other difficulties, and 4 ≡ nothing. Variable (13) `Consume` (most time consuming action) is graded as: 1 ≡ editing the changes, 2 ≡ understanding the code, and 3 ≡ removing errors. Variable (15) `Understand` (how well was the code understood) was graded as: 1 ≡ not very well, 2 ≡ fairly to moderately well, and 3 well to very well. Variable (16) `NotUndst` (any code not understood) is graded as: 1 ≡ none, 2 ≡ string class, and 3 ≡ certain C++ syntax. Variable (17) `CodeQual` (quality of code written) is graded as: 1 ≡ poorer quality, 2 ≡ similar quality, and 3 ≡ as good as or better quality. Variable (18) `Different` (do anything different next time) is graded as: 1 ≡ no, 2 ≡ use inheritance, 3 ≡ add comments, 4 ≡ be more sure of syntax, and 5 ≡ draw the inheritance hierarchy. Variable (19) `Learned` (learned anything) is graded as: 1 ≡ nothing, 2 ≡ inheritance, 3 ≡ OO is beneficial, and 4 ≡ other answers. Finally, variable (20) `Extra` (extra comments made) is graded as: 1 ≡ nothing, 2 ≡ mention of task being straightforward, 3 ≡ steep learning curve for OO,

	Induction Variables																		
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	A	44	12	n	5	2	y	1	1	2	1	1	b	3	1	3	2	1	1
2	A	37	0	n	3	1	y	1	1	2	4	1	c	2	1	3	1	1	2
3	A	31	0	y	3	2	y	1	1	3	4	1	a	3	2	3	1	2	4
4	A	99	0	n	12	1	n	-	-	-	-	-	-	-	-	-	-	-	-
5	B	27	0	y	1	2	y	1	1	3	4	1	a	3	1	3	1	1	4
6	A	100	45	n	37	1	y	2	3	5	2	3	a	2	1	1	2	4	1
7	A	64	0	y	5	1	y	2	3	7	2	3	a	2	1	3	1	1	4
8	B	-	0	y	25	1	y	3	3	10	2	3	a	1	3	1	1	4	4
9	A	-	0	y	10	3	n	-	-	-	-	-	-	-	-	-	-	-	-
10	A	49	0	y	7	1	y	1	2	6	2	3	b	2	1	3	1	2	1
11	B	-	0	y	21	1	y	1	3	5	3	3	a	2	2	1	1	4	1
12	A	38	0	y	12	2	n	-	-	-	-	-	-	-	-	-	-	-	-
13	B	47	0	y	6	3	y	1	1	3	2	1	a	3	1	3	1	4	1
14	B	25	0	y	4	3	y	1	1	2	2	1	c	3	1	3	1	4	1
15	B	32	0	y	2	1	y	1	1	1	3	1	b	3	1	3	1	1	1
16	A	60	0	n	12	1	y	1	1	2	2	3	a	3	1	2	1	4	1
17	B	36	0	y	6	3	y	1	1	3	2	3	a	3	1	3	1	2	4
18	A	85	0	n	17	1	n	-	-	-	-	-	-	-	-	-	-	-	-
19	B	58	0	y	11	1	y	1	2	2	3	3	c	3	1	3	1	4	1
20	A	56	0	y	7	1	y	1	1	6	3	1	a	2	1	1	1	3	1
21	A	35	0	n	2	1	y	1	1	2	2	1	c	3	1	3	2	1	1
22	B	36	0	y	2	1	y	2	1	2	2	1	a	3	3	2	1	4	1
23	B	49	0	y	8	1	y	2	1	4	2	3	c	2	1	2	1	2	1
24	B	52	0	y	9	1	y	2	1	3	1	3	d	2	1	2	1	2	1
25	A	41	0	y	12	2	y	1	1	1	4	3	a	3	1	3	1	1	1
26	A	-	0	y	24	1	y	2	3	4	2	3	a	2	1	2	1	1	3
27	B	31	0	y	9	1	y	2	1	5	2	1	a	3	1	2	1	4	1
28	B	41	0	y	13	1	y	1	1	2	2	3	a	3	1	3	1	4	4
29	B	29	0	y	1	1	n	-	-	-	-	-	-	-	-	-	-	-	-
30	B	102	0	y	11	1	y	2	2	3	2	3	a	2	1	3	1	4	4
31	A	-	0	y	15	1	y	3	3	7	2	3	a	2	1	2	1	4	1

Table D.2: Raw data for the second run of the first experiment

and 4 ≡ other answers.

D.7.2 Internal replication and second experiment

The logical groupings of data for variables in Table D.3 (the internal replication) are now explained. Variable (7) `Experience` (programming experience) is graded as: 1 ≡ a final year student, 2 ≡ a new graduate student, and 3 ≡ a PhD student. Variable (9) `InstTime` (time to understand the experimental instructions) is graded as above. Variable (10) `SynDiff` (difficulty with syntax) is graded as above. Variable (12) `Diff` (what caused the most difficulty) is graded as: 1 ≡ inheritance related difficulties, 2 ≡ syntax difficulties and compiler errors, 3 ≡ object construction and initialisation, 4 ≡ nothing, and 5 ≡ other answers. Variable (13) `Consume` (most time consuming action) is graded as: 1 ≡ editing and making changes, 2 ≡ understanding the code, and 3 ≡ removing errors. Variable (15) `Understand` (how well was the code understood) is graded as above. Variable (16) `NotUndst` (any code not understood) is graded as: 1 ≡ none, 2 ≡ string class, and 3 ≡ other replies. Variable (17) `CodeQual` (quality of

	Induction Variables																		
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	B	41	0	y	19	1	y	1	3	3	2	3	d	2	1	3	1	1	2
2	B	27	0	n	3	2	y	1	1	4	4	1	d	2	1	3	1	1	2
3	A	32	0	y	5	2	y	2	1	3	3	3	c	2	1	2	1	2	4
4	A	29	0	y	2	1	y	1	1	1	4	1	d	3	1	3	1	1	2
5	A	-	0	y	37	3	y	3	3	4	3	3	b	2	3	1	4	5	1
6	B	97	0	y	30	2	y	3	3	5	5	2	b	2	1	3	3	2	1
7	A	19	0	y	2	2	y	1	1	2	5	1	c	2	1	3	1	5	1
8	A	51	0	y	5	2	y	1	2	2	3	3	c	2	1	3	4	3	1
9	A	35	0	y	-	2	y	1	2	3	4	1	d	2	2	3	1	1	1
10	B	51	0	y	12	1	y	1	2	2	3	3	b	3	1	2	1	5	1
11	A	77	0	y	27	2	y	3	1	5	2	3	c	2	1	2	1	5	1
12	A	44	0	y	6	2	y	2	3	5	1	3	a	3	3	2	1	4	1
13	A	28	0	y	4	1	y	1	3	1	3	3	d	2	1	3	1	1	1
14	B	63	0	n	18	2	y	1	3	3	5	3	d	2	1	3	2	2	2
15	B	63	0	y	10	1	y	1	3	4	3	3	a	2	1	2	1	4	4
16	B	59	0	y	21	2	y	2	1	2	2	3	b	2	1	1	1	2	4
17	A	14	0	y	2	2	y	1	1	3	1	1	c	2	1	2	1	3	3
18	B	31	0	y	5	2	y	2	2	3	3	1	b	3	3	1	1	4	1
19	A	22	0	y	2	1	y	1	1	2	2	1	d	3	1	2	1	1	2
20	B	22	0	y	2	1	y	1	1	1	3	1	c	3	1	3	1	3	2
21	A	52	0	y	10	1	y	1	3	3	2	3	a	3	1	3	1	1	1
22	B	35	0	y	4	2	y	2	2	2	4	1	a	3	1	3	1	3	1
23	B	44	0	y	3	2	y	2	1	2	3	1	d	3	1	2	1	3	1
24	A	26	0	y	4	2	y	1	1	1	4	1	c	3	1	3	1	1	1
25	B	38	0	y	5	1	y	1	2	4	3	1	b	2	1	3	1	3	1
26	A	29	0	y	7	2	y	1	2	3	5	1	b	3	1	3	1	3	1
27	B	25	0	y	1	2	y	1	1	2	1	1	d	2	2	3	1	2	4
28	A	-	0	y	15	3	y	3	3	9	5	3	a	1	1	1	1	5	1
29	B	49	0	n	5	2	y	2	3	2	2	3	a	2	1	3	2	1	1
30	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
31	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table D.3: Raw data for the internal replication

code written) is graded as above. Variable (18) `Different` (do anything different next time) is graded as: 1 ≡ no, 2 ≡ use inheritance, 3 ≡ read instructions properly, and 4 ≡ concentrate on object construction. Variable (19) `Learned` (learned anything) is graded as: 1 ≡ nothing, 2 ≡ inheritance benefits, 3 ≡ extensibility of OO code, 4 ≡ OO is beneficial, 5 ≡ other answers. Finally, variable (20) `Extra` (any extra comments) is graded as: 1 ≡ nothing, 2 ≡ modification was straight forward, 3 ≡ OO is good to work with, and 4 ≡ other answers.

The logical grouping of attributes in Table D.4 (the second experiment) are the same as above except for the following, variable (12) `Diff` (what caused the most difficulty) is graded as: 1 ≡ inheritance related difficulties, 2 ≡ syntax difficulties and compiler errors, 3 ≡ choosing which class to specialize from, 4 ≡ part of the modification, and 5 ≡ other answers. Variable (13) `Consume` (most time consuming action) is graded as: 1 ≡ editing and making changes, 2 ≡ understanding the code, 3 ≡ removing errors, and 4 ≡ inheritance related difficulties. Variable (16) `NotUndst` (any code not understood) is graded as: 1 ≡ none, 2 ≡ string class, 3 ≡ other replies, and 4 ≡

	Induction Variables																		
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	B	33	0	y	-	1	y	2	1	2	3	2	d	3	1	3	1	1	3
2	B	33	0	y	5	2	y	1	2	7	4	3	d	2	1	3	4	2	1
3	A	77	24	n	15	2	y	3	3	4	1	3	c	2	1	2	2	4	1
4	A	65	20	n	13	1	y	3	3	6	1	3	a	3	1	3	2	1	1
5	A	31	0	n	8	3	y	2	1	1	5	3	a	2	1	2	1	3	1
6	B	90	0	y	17	2	y	3	3	7	1	2	b	2	3	2	4	2	1
7	A	15	0	n	1	2	y	2	2	3	5	1	c	2	4	3	1	2	1
8	A	29	0	n	7	2	y	1	1	1	5	3	c	2	1	3	4	4	1
9	A	40	0	n	7	2	y	1	2	5	4	1	c	2	4	3	1	4	1
10	B	65	0	y	14	1	y	1	2	4	4	4	b	3	1	3	3	2	1
11	A	47	0	n	7	2	y	3	2	4	5	2	a	2	1	2	1	4	1
12	A	57	0	n	8	2	y	2	1	6	4	1	a	2	1	2	2	4	1
13	A	92	0	y	31	1	y	1	2	3	5	2	d	2	1	3	1	4	1
14	B	34	0	y	11	2	y	1	2	3	5	1	c	2	1	3	1	1	1
15	B	65	0	y	5	1	y	3	2	3	5	2	a	3	1	3	1	3	1
16	B	53	0	y	24	2	y	2	1	4	4	2	b	2	1	1	1	2	1
17	A	73	0	y	11	2	y	1	3	7	1	4	a	2	3	3	3	4	1
18	B	34	0	y	3	2	y	3	1	4	1	1	a	2	3	3	4	3	1
19	A	44	0	y	7	1	y	2	2	5	1	2	d	2	1	3	4	4	1
20	B	53	0	y	10	1	y	2	3	3	2	3	d	3	4	3	1	3	1
21	A	-	0	y	2	1	y	1	1	6	1	1	d	2	1	3	1	3	1
22	B	50	0	y	9	2	y	1	1	3	4	3	c	3	1	3	3	3	1
23	B	48	0	y	5	2	y	2	1	3	1	1	d	3	1	2	1	3	1
24	A	34	0	n	3	2	y	1	1	3	5	1	c	3	1	3	2	4	1
25	B	36	0	y	4	1	y	1	1	7	3	2	b	2	4	3	1	1	3
26	A	29	0	n	5	2	y	1	1	1	2	1	d	3	1	3	2	3	1
27	B	76	0	y	19	2	y	2	3	6	1	4	b	2	1	2	4	3	1
28	A	73	0	n	7	3	y	2	2	5	5	1	c	1	1	2	4	4	1
29	B	115	0	y	19	2	y	1	1	7	3	3	b	2	1	3	1	3	3
30	B	40	0	y	6	1	y	2	1	3	3	1	a	3	1	3	1	4	2
31	A	46	0	n	10	2	y	3	2	3	3	1	c	2	3	2	2	4	1

Table D.4: Raw data for the second experiment

everything but the relevant parts. Variable (18) `Different` (do anything different next time) is graded as: 1 ≡ no, 2 ≡ use inheritance, 3 ≡ draw the inheritance hierarchy, and 4 ≡ other answers. Variable (19) `Learned` (learned anything) is graded as: 1 ≡ nothing, 2 ≡ inheritance hierarchies can be complex, 3 ≡ use of inheritance/inheritance has benefits, 4 ≡ other answers. And variable (20) `Extra` (extra comments made) is graded as: 1 ≡ nothing, 2 ≡ modification was straight forward, 3 ≡ deciding which class to specialize from was difficult.

Appendix E

Miscellaneous

E.1 External replication materials

E.1.1 Pilot debriefing results

Subjects were informally debriefed after the pilot study for the external replication. Their substantive comments (paraphrased here) on the pretest and experiment were as follows:

Pretest:

- An unrealistic problem but useful for Turbo Pascal practice.
- Use of a fixed array for file processing was unnatural.

Experiment:

- The pretest taught you the semantics of the task (in pretest, use disk seek read and write pairs and replace with array assignment statements while in the experiment use array assignment statements and replace these with disk seek read and write pairs).
- No intellectual capacity needed to perform changes (in either monolithic or modular programs).
- In the modular program, there is an instruction to delete the global array variable *InventoryArray* but this happens to be the last variable declaration which is immediately followed by the comment that the following procedures are the only

ones that ever use this array: as a consequence I no longer have to consider the program, just looking at the four procedures.

- In the monolithic program, the task turns into a global search and replace operation and as such is unrealistic because I can't use the editing environment and feel at a disadvantage as I am having to manually search through listing for all the changes.
- In the modular program, unhappy that arrays are passed as global structures instead of being passed as parameters: means that the task is much easier than it should be.
- After first pass, editing and compiling wrapped into logic phase.
- Suggest highlighting name of file.
- If you make a mistake, only then do you have to try and understand what is going on.
- You have to remember to restore the test data files properly if testing reveals you have made a mistake otherwise you can end up taking even more time.
- Program layout and commentary could be improved.
- I think I had written down all the changes correctly on paper but had incorrectly typed in one of the changes.

Several of these comments corroborate some of the criticisms presented in Chapter 4, Section 4.2.2 and some can be viewed as identifying additional weaknesses of Korson's study. Note that there exists similarity between some of the comments above and some of those made on debriefing questionnaires by replication subjects.

E.1.2 Debriefing questionnaire

Thank you for participating. Before you leave, please give a few personal details and your comments.

Personal details

Name:
Age:
Sex:
Position (e.g., 2nd year CS):
Qualifications if you are a graduate (e.g., BSc. Comp. Sci. 2(i)):

Comments

1. Did you find the task easy?

2. Did you make good use of the editor?

3. What caused you the most difficulty?

4. How well do you understand the code?

5. Have you learned anything? If so, what?

Any other comments?

	Induction Variables																		
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
A	15	4	1	2	22	18	mod	237	26	m	rs	y	y	3	3	1	1	0	n
B	14	13	7	4	38	25	mod	58	22	m	ra	y	y	1	1	2	5	1	n
C	16	6	1	10	33	27	mod	98	21	m	cs4	n	y	2	2	1	1	2	n
D	22	3	1	20	46	43	mod	105	22	m	cs4	n	y	3	0	3	5	0	n
E	39	14	3	14	70	56	mod	116	20	m	cs3	n	y	1	3	3	1	0	n
F	34	7	19	5	65	58	mod	144	33	m	rs	y	y	2	2	3	2	2	n
G	31	2	29	1	63	61	mod	69	27	m	ra	n	n	3	1	3	3	5	n
H	46	16	25	25	112	96	mod	176	23	m	rs	n	y	3	2	3	2	2	y
I	15	15	4	5	39	24	mon	180	20	m	cs4	y	y	2	0	4	2	3	n
J	32	30	6	3	71	41	mon	333	19	f	cs2	y	n	3	0	4	2	1	n
K	38	14	2	4	58	44	mon	679	20	m	cs3	y	y	2	2	3	2	1	n
L	44	21	1	4	70	49	mon	652	22	m	cs4	y	n	2	0	1	4	0	n
M	28	30	11	11	80	50	mon	562	22	m	cs4	y	n	2	1	3	4	1	y
N	36	47	22	1	106	59	mon	429	33	m	rs	y	n	2	1	1	2	3	n
O	29	27	3	29	88	61	mon	445	22	m	cs2	y	y	3	1	3	2	0	y
P	55	26	2	36	119	93	mon	661	26	m	ra	n	y	3	0	1	2	0	y
Q	38	22	15	58	133	111	mon	644	33	m	cs2	y	n	3	3	2	2	1	y

Table E.1: Raw data for external replication study

E.1.3 Inductive analysis raw data

Table E.1 displays the data in its raw form before any manipulation took place. Similar replies given to each question on the debriefing questionnaire were grouped so that they had the same logical value (as described in Chapter 4).

E.2 Basili's experimentation framework paradigm

I. Definition					
Motivation	Object	Purpose	Perspective	Domain	Scope
Understand	Product	Characterize	Developer	Programmer	Single Project
Assess	Process	Evaluate	Modifier	Program/project	Multi-project
Manage	Model	Predict	Maintainer		Replicated Project
Engineer	Metric	Motivate	Project manager		Blocked subject-project
Learn	Theory		Corporate manager		
Improve			Customer		
Validate			User		
Assure			Researcher		

II. Planning		
Design	Criteria	Measurement
Experimental designs	Direct reflections of cost/quality	Metric definition
Incomplete Block	Cost	Goal-question-metric
Completely randomised	Errors	Factor-criteria-metric
Randomised block	Changes	Metric validation
Fractional factorial	Reliability	Data Collection
Multivariate analysis	Correctness	Automatability
Correlation	Indirect reflections of cost/quality	Form design and test
Factor analysis	Data coupling	Objective vs. subjective
Regression	Information visibility	Level of measurement
Statistical models	Programmer comprehension	Nominal/classificatory
Non-parametric	Execution coverage	Ordinal/ranking
Sampling	Size	Interval
	Complexity	Ratio

III. Operation		
Preparation	Execution	Analysis
Pilot study	Data collection	Quantitative vs. qualitative
	Data validation	Preliminary data analysis
		Plots and histograms
		Model assumptions
		Primary data analysis
		Model applications

IV. Interpretation		
Interpretation context	Extrapolation	Impact
Statistical framework	Sample representativeness	Visibility
Study purpose		Replication
Field of research		Application

Table E.2: The experimentation framework paradigm [Basili *et al.*, 1986]