An Empirical Investigation of Issues Relating to Software Immigrants

by

Alistair J. Hutton

Presented to the Faculty of Information and Mathematical Sciences
Of the University of Glasgow
For the degree of Ph.D by Research

2008

Abstract

This thesis focuses on the issue of people in software maintenance and, in particular, on software immigrants – developers who are joining maintenance teams to work with large unfamiliar software systems. By means of a structured literature review this thesis identifies a lack of empirical literature in Software Maintenance in general and an even more distinct lack of papers examining the role of People in Software Maintenance. Whilst there is existing work examining what maintenance programmers do the vast majority of it is from a managerial perspective, looking at the goals of maintenance programers rather than their day-to-day activities. To help remedy this gap in the research a series of interviews with maintenance programmers were undertaken across a variety of different companies. Four key results were identified: maintainers specialise; companies do not provide adequate system training; external sources of information about the system are not guaranteed to be available; even when they are available they are not considered trustworthy. These results combine together to form a very challenging picture for software immigrants. Software immigrants are maintainers who are new to working with a system, although they are not normally new to programming. Although there is literature on software immigrants and the activities they undertake, there is no comparative literature. That is, literature that examines and compares different ways for software immigrants to learn about the system they have to maintain. Furthermore, a common feature of software immigrants learning patterns is the existence and use of mentors to impart system knowledge. However, as the interviews show, often mentors are not available which makes examining alternative ways of building a software immigrants level-of-understanding about the system they must maintain all the more important. As a result the final piece of work in this thesis is the design, running and results of a controlled laboratory experiment comparing different, work based, approaches to developing a level-of-understanding about a system. Two approaches were compared, one where subjects actively worked and altered the code while a second group took a passive 'hands-off' approach. The end result showed no difference in the level-of-understanding gained between the subjects who performed the active task and those that performed the passive task. This means that there is no benefit to taking a hands-off approach to building a level-of-understanding about new code in the hostile environment identified from the literature and interviews and software immigrants should start working with the code, fulfilling maintenance requests as soon as possible.

Acknowledgements

I would like to thank my supervisor, Professor Ray Welland, who has been a constant source of advice and help throughout my PhD. Also Phil Gray's difficult questions have helped me shape this thesis. My girlfriend, Clare Sheehy Skeffington has worked countless hours correcting my never ending stream of grammatical mistakes, poorly punctuated sentences and made up words. Dr Stephen Draper and Rebbeca Smith where vital in providing debate and advice in the construction of my experiment. Professor John McColl was invaluabe in pointing me in the right direction when it came to the experimental statistics. Finally I'd like to acknowledge the 52 students who took part in my experiment and the 11 maintenance programmers who agreed to be interviewed by me. Without any of these people my thesis would not have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Although the exact figure varies, current research [40] states that the majority of a software system's lifetime is spent in the maintenance phase. However, relative to this, the least amount of research in Software Engineering has been done on the topic of Software Maintenance. Furthermore, many still reference empirical research from 30 years ago [45] as if it is current fact [68, 78, 61], despite the current state-of-practice having moved on. This makes the examination of Software Maintenance of vital importance in the field of Software Engineering as a whole.

Before informed decisions can be made about how to change, augment or otherwise aid the Software Maintenance process, empirical data must be gathered on what happens when Software Maintenance takes place. Given that Software Maintenance is a field that is driven by the existence of actual systems needing to be maintained, the empirical data must be continually updated or reduce in relevance over time. New approaches must be compared, not just with other advocative research, but also against current practices to discern their utility.

However, like Software Engineering and Computing Science as a whole [86, 76], Software Maintenance has a demonstrable lack of papers containing either empirically based work or empirical validation of work. Furthermore, examining the *subject* of the work shows that the majority of Software Maintenance research focuses on the Product and Process elements of Software Maintenance, with papers examining People forming less than a tenth of all work published in the mainstream literature. This is a surprising result given the vast reported differences in the ability of professional programmers [59]. In some cases programmers were up up to 28 times more efficient than their peers. This seems to show the great importance of People in the overall picture of Software Maintenance yet, as stated, there is a lack of research focused on them. Particularly lacking is research examining what maintainers do on a day-to-day basis: information that would surely be useful in trying to formulate research goals to aid maintainers. Whilst there *is* data on what type of work maintainers produce, this is a managerial overview of maintainers and does not deal in the subtle complexities and practicalities that are involved in actually performing maintenance.

Although there is a need to increase the general body of knowledge about maintainers, there is also scope to address specific areas. The topic chosen for this thesis is that of software immigrants (those brought in to help maintain systems as described below in section 1.1.2). Software immigrants have a need to develop a level-of-understanding about the system they have to maintain, yet they come into an environment that lacks any useful sources of information. Although there is assorted literature of different approaches to aiding software immigrants gain a level-of-understanding about a sub-system, there is a lack of *comparative* literature that examines the utility of different approaches.

As such, after providing a solid foundation for the work, this thesis will present the work of performing some comparative analysis of different approaches to software immigrants building a level-of-understanding of a system.

1.1 Definitions

1.1.1 Hierarchy of Knowledge

I classify a programmer's knowledge according to the pyramid shown in figure 1.1. Technical knowledge is the command and understanding of programming languages, databases and related general programming knowledge. Domain knowledge is knowledge about types of software systems and the problem areas they are addressing, for example, internet banking systems, and how one would be designed and implemented. System knowledge is knowledge about a particular software system that is in a domain, for example, Company X's internet banking system. Sub-system knowledge is specific, code level understanding of a particular facet of a particular system, for example, the transaction processing part of Company X's internet banking system.

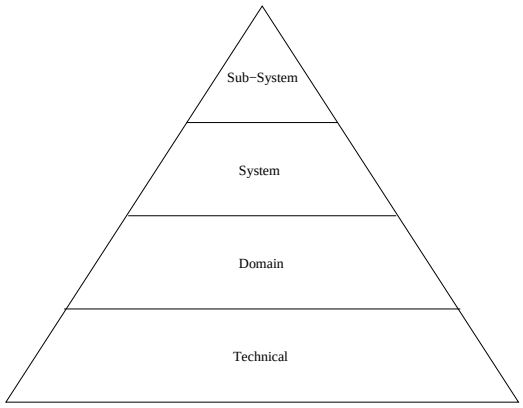

Figure 1.1: The Knowledge Pyramid

1.1.2 Software Immigrants

Software Immigrants are programmers who have just joined a team working on an established software system. This term was first defined by Sim and Holt [63] in their examination of how software immigrants learn about their new job, and is examined in

greater depth in chapter 4. Software immigrants are not synonymous with novice programmers in that they can have many years of professional programming experience before being brought to work with the current system.

Software immigrants will, for the most part, have firm technical knowledge. Some will be familiar with the domain they are working in and will be able to carry over domain knowledge. A few will have previously been working on a different part of the system and so will be able to carry over system knowledge. No immigrant, however, will have sufficient sub-system knowledge to be considered an expert on the sub-system. It is in aiding the development of the software immigrants' level-of-understanding about a sub-system that is the specific topic investigated by this thesis.

1.1.3 Level-of-Understanding

The term *Program Understanding* (and the interchangeably used term *Program Comprehension*) is overloaded, which results in confusion for even experts in the field. Program Understanding is used to refer to two related but conceptually very different ideas. The first is the modelling and examination of what happens in a programmers mind *as they build up knowledge about a piece of code*. In this sense, Program Understanding is referring to a process. The other topic is examining *what a programmer knows after a period of time working with a program*: the end result rather than the process of how the knowledge was obtained. So for example, the experiments of Daly et al. [17] and von Mayrhauser et al [81] are both referred to as program understanding experiments, even though they are measuring entirely different things. Therefore, to criticise the Daly et al. experiment on the grounds that it fails to analyse the thought process of the subjects is a nonsequitur. The literature seems to have more papers using of the term Program Understanding to mean the first idea. To avoid confusion I will follow this convention. For the second concept I will use the term *level-of-understanding*.

1.2 Thesis Statement

The environment in which software immigrants have to work is often more challenging than common belief imagines. Limited empirical work does not show what are the best approaches for software immigrants to take. It can be shown that the best way for software immigrants to build a level-of-understanding about a system is to start working with the code as soon as possible.

1.2.1 Measures of success

To test the thesis this dissertation will present a range of work which can be evaluated against the following measures of success:

- Present evidence on the current state of mainstream software maintenance research in relation to empirical research to show a gap in the literature
- Perform general work to help close the gap

- Identify unresearched problems of software immigrants
- Compare and contrast different approaches for software immigrants to develop a level-of-understanding about a system

This thesis is founded upon the assumption that there is insufficient empirical work being performed in Software Maintenance. To validate this assumption, evidence is provided by means of a systematic review of research trends in Software Maintenance. After demonstrating a lack of literature, research that fits in that gap is performed, to use as a basis for further research. This general work suggests a specific line of specialised enquiry which itself needs to be further researched to understand its scope and problems. Finally, having identified and specified a specific area of research, the final piece of research, which goes to justifying the thesis statement, is performed by the means of a quantitative experiment.

1.3 Thesis Structure

Chapter 2 contains an analysis of the classic pieces of software maintenance research while drawing attention to the gap of knowledge around People in software maintenance. A systematic literature review is presented to formally show the lack of empirical and People oriented papers in mainstream software maintenance research.

Chapter 3 presents the results of interviewing a series of maintenance programmers, to try and help remedy the lack of empirical papers about maintainers. Highlighted in this chapter are the interesting research questions surrounding software immigrants.

Chapter 4 is a second literature review which focuses purely on the research published on software immigrants, comparing it with the results of the interviews.

Chapter 5 presents the basis for designing, running and measuring the results of Software Engineering experiment dealing with level-of-understanding issues.

Chapter 6 shows the design, implementation and results of a laboratory experiment to measure the differences in programmers' level-of-understanding from taking different learning approaches to an unfamiliar system while.

Chapter 7 presents further possible work which could use this thesis as a basis.

Chapter 8 presents a summary and conclusions of the thesis.

1.4 Contribution

The contribution of this thesis can be split into three sections:

- Structured Literature Review — The research trends review of the literature is a unique piece of work which bears comparison to similar studies in the domains of Software Engineering and Computing Science. It paints a clear picture of the state

of empirical research in software maintenance and it motivates the remainder of the work in the thesis.

- Maintenance Programmer Interviews — The interviews are a partial replication of the work of Singer [64] which showed broadly similar results. Due to the confirmation of results of the replicated sections, this allows more confidence in the generalisability of the results in the non-replicated section of the interviews. The interviews show that the potential environment for software immigrants can often be harsher than common belief and research in the field assumes.

- Finally, there is the experiment. With extensive reference to established experimental guidelines it is shown to be a rigorous and robust design which uses the appropriate statistics of survival analysis to calculate the quantitative results. These are the necessary results to support the thesis statement of section 1.2. The design and main results of the experiment have been presented at Empirical Assessment of Software Engineering 2007 as Hutton and Welland [33].

Chapter 2

Literature Review

2.1 Introduction

This chapter is divided into two main sections. The first starts with a brief overview of what constitutes Software Maintenance, and then gives a thorough examination of the two key pieces of research in the field: the Lientz, Swanson (& Tompkins) surveys and Lehman's Laws of Evolution. The second section poses two questions about the nature of Software Maintenance research: whether there is enough empirical work and if there is an equal proportion of work examining People in Software Maintenance as there is examining the Product and Process aspects. These two questions are answered by means of a systematic literature review of mainstream Software Maintenance publications.

2.2 What is Software Maintenance

Software Maintenance is the continued change of a computer program after it has been released to users. The term covers not just the fixing of bugs and errors but the adaptation of the program to a changing environment and the accommodation of requests for improvements from both within and outside the maintenance team.

2.2.1 What are the Components of Software Maintenance

Software Engineering is the study of People, Processes and Products (the 3Ps) [14]. The study of Software Maintenance, being a subset of Software Engineering, is therefore also a study of People, Processes and Products. 'People' covers everyone involved in the maintenance process: managers, programmers, testers, deployment specialists and customers. 'Processes' covers not only the formal processes that maintainers undertake (e.g. Waterfall, Spiral model, XP etc.) but also tool usage and programming techniques. 'Products' covers the software system as well as all artefacts the maintainers use while carrying out processes, such as documentation, maintenance request forms, and system logs.

The two classic pieces of research in Software Maintenance are the Lientz, Swanson (& Tompkins) surveys of maintenance work [45, 44] and Lehman's Laws of Software Evolution [43]. The various papers that make up these two pieces of research are cited by hundreds of

other papers, and are the only two sources referenced in the software maintenance chapters of the major Software Engineering text books [68, 78, 61]. The Lientz & Swanson survey covers Processes and Product issues, and is cited by well over 350[1] separate papers, while Lehman's Laws are primarily focused on Product issues with some material on Processes, and is also cited by more than 350 papers. However, there is no 'classic' study, cited by hundreds of papers, that deals with the People issues in Software Maintenance.

2.3 The Lientz, Swanson (& Tompkins) Surveys

There are two classic pieces of research in the field, which both take a management level view of maintenance activity: i.e. they study what managers think is occurring during maintenance. These are the Lientz, Swanson & Tompkins survey of 69 companies from various sectors published in 1978 [45] (to be known as LST for brevity), analysis of which made up the bulk of Tompkins PhD thesis, and the Lientz & Swanson larger scale follow up survey of 467 companies, published in 1980 using the same methodology [44] (to be known as LS). These surveys uncovered a large amount of information about how maintenance was regarded at the time. Amongst all the data there are three key measures: the percentage of time spent working on a system which is dedicated to maintenance; the distribution of work over different types of maintenance; and a ranking of severity of around 25 different problem areas. The first two results are widely referenced, while the third receives rather less attention, despite the interesting information that it contains. Nosek & Palvia (NP) performed a replication of the LS study with 51 companies in the late 1980s [51] finding broadly similar results. For ease of reference, these three surveys, using as they do identical methodology, will be referred to collectively as the *original surveys*.

2.3.1 Percentage of Time Dedicated to Maintenance

One of the key results of the surveys is the proportion of time that is spent maintaining systems rather than performing new development. The figures for this result in the three studies are given in table 2.1. The Accuracy of Data column represents the percentage of respondents who stated that their response was "Reasonably Accurate, based on good data". The other two categories were "A rough estimate, based on minimal data" and "A best guess, not based on any data".

The NP study shows that in the ten years between its survey and the LS survey, there was a statistically significant increase in the proportion of time being spent on maintenance. The LST study notes that their result of 48% is low in comparison to contemporary estimates. Yet, at the same time, they also report that over 20% of their respondents allocated 85% of their effort towards maintenance. This is a very large deviation from the average, and in fact means that the average for the other 80% of respondents would be around 38%, thus putting some of the respondents well below even the lowest estimates for effort in software maintenance. The LS survey breaks down the average time by industrial sector, which results in a range of proportions from 65.26% down to 26.25%, with a standard

[1] Citation numbers obtained by use of Google Scholar http://scholar.google.com

deviation of around 23 points: this would leave some groups doing almost no maintenance at all. The industrial sectors were split by a manufacturing/service industry distinction, with metal fabrication, paper working and petroleum based industries being the largest contingents of the Manufacturing industries and insurance, banking and governmental departments being the largest Service industry groups. In general, Service industries performed more maintenance than Industrial industries, with the the top groups being data processing services, data processing equipment manufacturers, investment, banking and insurance with a rage of percentages from 65% to 55%, while the bottom groups were 'other service industries', printing/publishing, chemical/allied, textiles and consultancy with a range of 41% to 26%. Koskinen's 2003 review of the literature on the topic leads him to conclude that modern average maintenance effort has now easily risen to the area of 70%, a view also shared by Pigoski [53]. This suggests that the figures produced in the original surveys for the proportion of time spent on maintenance have become dated and should only be used as a historical context to show a rising trend.

Study	Percentage	Accuracy of Data
LST	48.0	82.6
LS	48.8	52
NP	58	42

Table 2.1: Percentage of Department Time Spent on Maintenance (with accuracy of answers)

2.3.2 Work Distribution over Maintenance Types

Probably the most widely quoted result for the LST and LS studies is the distribution of work amongst four categories identified by Swanson: Perfective, Adaptive, Corrective and Other [72]. Perfective maintenance is defined as improvements to documentation, improving the efficiency of the system for non user requested reasons, and adding user enhancements. Adaptive maintenance is altering the system so it can accommodate change to the environment in which it operates. Corrective maintenance is routine debugging and emergency fixes. The 'Other' category covers tasks that result in change to the system that do not fit into the first three categories.

The Nature of Swanson's Categorisations

While the figures for these studies are widely quoted, there appears to be almost as widespread as confusion as to exactly what the figures are. Of the major Software Engineering textbooks, Van Vliet [78] refers to the LS survey, but changes the figures by whole percentage points. Schach [61] references the LST paper for the figures rather than the larger scale LS study of the same period. Sommerville [68] references the LS and NP studies but gives the numbers from LST, whilst folding the Other category into the figure for Perfective maintenance.

These textbooks all focus on one particular aspect of the results: that fixing bugs is a small proportion of what maintenance programmers do, while changes to functionality (Adaptive and Perfective maintenance) are more important. However, I feel that this

ignores the original tone of the Swanson categorisations, which make it clear that Corrective and Adaptive maintenance should be considered together as *unavoidable* sources of maintenance whilst Perfective maintenance represents *voluntary* reasons to make changes. I think that, therefore, the more useful interpretation of the figures is that the LS and NP surveys show that unavoidable sources of change are almost as common as voluntary forms of change. I think that this is also the more important result from a managerial point of view, as a knowledge of the relative amount of unavoidable work is of more use for determining how well the department is performing than focusing on the amount of Corrective maintenance performed. This is particularly pertinent considering that the figure for Corrective maintenance seems to be highly dependent on system characteristics (see the following section).

Interpreting changes between the LST/LS studies and the NP study is made difficult since NP only presented partial data. They provide all the information needed to calculate the Corrective maintenance category but exclude three subcategories needed for calculating the Perfective and Adaptive percentages. As can be seen there is an increase in the figure for Corrective maintenance over the three surveys, a difference of 35% between the LST and NP studies.

Study	Perfective	Adaptive	Corrective	Other	Accuracy of Data
LST	60.3	18.2	17.4	4.1	49.3
LS	51.3	23.6	21.7	3.4	30.0
NP	42+	17+	23	NA	NA

Table 2.2: Percentage of Work Distribution by Maintenance Category (with accuracy of answers)

Empirical Studies of Work Categorisation Based on Primary Sources

The data from the LS, LST and NP surveys were obtained by the same method: a questionnaire administered to the department managers who based their answers on not particularly good data. It would be interesting to determine if these figures would be validated by examining work categorisations in another, more direct, way. Three studies offer this opportunity: Schach et al. [60], Mockus & Votta [47], and Abran and Nguyenkim [2] which will be referred to as SEA, MV & AN respectively.

SEA took the CVS history for three products (**gcc**, the **Linux** kernel and **RTP**), and hand classified every change to the system at both code and module level into Perfective, Adaptive, Corrective or Other. The MV study started by producing a piece of software that automatically classified change-log entries in a company's CVS system, and then used it to classify the work performed on a software system that the company maintained into the categories Adaptive, Corrective, Perfective, Inspection and Other. The AN study had access to two years' worth of maintenance request and fulfilment reports from a company covering several different systems. These reports were then classified into Perfective, Adaptive, Corrective and a 4th category, User Support. They do not include an Other category.

Both AN and MV use the Swanson classifications differently from the way they were originally used. This is understandable given that the original Swanson classification does not include User Enhancements, however, the LST study makes it quite clear that User Enhancements are considered Perfective maintenance, saying specifically: "*Perfective: user enhancement, improved documentation, recoding for computational efficiency*". The eight original sub-classifications that the original surveys use are: Emergency Program Fixes, Routine Debugging, Change to input/output, Change to hardware/software, User Enhancements, Programming Documentation Improvements, Computational Efficiency Improvements, Other. Whilst the AN study states that they are using the Swanson categorisations, including a paraphrasing of the Adaptive category, their presentation of the results does not match with this claim. At first glance (see table 2.3) their restatement of the figures from the LS study looks like they have moved the User Enhancement sub-classification from Perfective to Adaptive. However, they have done more than that: they have also moved "Accommodation of change due to hardware and software change" to the Perfective category, something that is totally unsupported either by Swanson's original classification or AN's own restatement of the category. The MV study also redefines terms, making Adaptive consist of User Enhancements as well as the Swanson sub-categories. Furthermore, along with Perfective, Adaptive, Corrective and Other they introduce a 5th category, Inspection, which covers changes to the code which were made due to code inspections being performed. They state that these changes cover both Perfective and Corrective changes.

Perfective	Adaptive	Corrective	User Support
16	59	22	3

Table 2.3: Abran and Nguyenkim Restatement of Lientz and Swanson Study

Of greatest concern when it comes to comparing the results of the AN study to the original surveys is their relabelling of the original surveys' category Other as User Support. The AN Study defines User Support as "work orders [which] do not request changes but only information on the software components". User Support requests do not in and of themselves result in any work being done to the system. The original surveys did not measure this, nor did they set out to measure it [73]. As a result, comparing, say, the AN percentage for Corrective maintenance against the LS figure should not be done directly as this User Support category does not exist in the original surveys' figures. This additional category will skew the other numbers in a downward direction, so I have attempted to re-normalise the data to aid comparison with the other studies. I have done this by removing the User Support category from the figures and then recalculating what proportion of work each of the remaining categories contribute. In contrast to this, the SEA survey states that it will use the Swanson categorisations as used in the original surveys, and does so.

Analysis of Differences

Given the difficulty of comparing the AN and MV figures to the original surveys' figures, I have used my suitably re-normalised figures for the AN study to provide a comparison of Corrective with non-Corrective maintenance over all the studies in table 2.4. This is

the only direct comparison that can be made between the surveys. As can be seen, the three primary source studies all produce a larger figure for Corrective maintenance than the original studies, and, in the case of SEA, massively so. There are several possible reasons for this. The LS study noted a correlation between system size and the amount of Corrective maintenance performed, in that the larger the system was, the more Corrective maintenance occurred. The average LS system was 53,000 lines of code. The average NP system was 204,000 lines of code. In SEA, **gcc** is 850,000 lines of code, the **Linux** kernel is around 1.8 million lines of code, while **RTP** is only 12,000 lines of code. **RTP** was the system in the SEA study that most closely matched the LS survey, but the results were still very significantly different (Adaptive: 13.8%, Corrective 42.8%, Perfective: 26.8%, Other 16.7%). The MV system studied was two million lines of code. The AN study examines four major programs (of over one million lines of code) and a selection (total figure unreported) of packages and small applications. The small applications and packages had a (non re-normalised[2]) Corrective maintenance figure of 18% as opposed to the large applications' Corrective maintenance figure of 40%. This suggests, as the LS study notes and the MV study states, that vastly increased system size results in a shifting of the work proportions, increasing Corrective maintenance.

Another possibility (also briefly touched upon by the AN study) is that certain types of software could have their own maintenance profiles, which are separate from the system quality. The top types of systems surveyed by LS were Payroll Systems, Order Entry and Bill & Invoicing. These types of systems tend to run in a predictable manner. **gcc** and **Linux**, on the other hand, are both highly configurable pieces of software which will receive extensive testing of corner cases in their day-to-day operation. Given the entirely different characteristics of an operating system kernel and Payroll systems, it seems reasonable to suggest that the type of maintenance work they might generate (both voluntary and unavoidable) would be different. In the case of **Linux** and **gcc**, this would result in a far higher figure for Corrective maintenance, but this should not cause concern as the higher figure would be expected. These system-specific signatures could have been hidden by the averaging of results in the original surveys, in a similar manner to the way the LST average maintenance effort of 48% hides the departments with a maintenance effort of 85%+. None of the original studies provide the standard deviations for their work proportions, and the figures are also produced for the department as a whole rather than individual programs, making it impossible to definitively determine if this is the case.

Study	Corrective Percentage	Non-Corrective Percentage
LST	17.4	82.6
LS	21.7	79.3
NP	23	77
AN	26	74
MV	33.5	66.5
SEA	70.1	29.9

Table 2.4: Corrective vs Non-Corrective

[2]The AN study lacks the data to accurately re-normalise these figures. However, the figures do not need to be re-normalised in order to demonstrate the differences that are shown within the study itself.

Definitions

Given the confusion over the naming and attribution of work categories, I feel it is incumbent on me to make clear what I consider the various categories to be. I would like to use the Munro categorisation [48], which consists of four categories: Perfective, Adaptive, Corrective and Preventative. In the Munro classifications, Perfective maintenance consists solely of User Enhancement; Adaptive and Corrective are exactly the same as the Swanson definitions; and Preventative maintenance consists of Swanson's original Perfective category: "*Performance Improvement, Documentation Updating and Code Structure Improvement*". The three best features of such a classification system are that:

- it maintains the spirit of Swanson's original classification by allowing the work to be split into unavoidable and voluntary change;
- it stops User Enhancements dominating the other categories that make up Swanson's Perfective maintenance;
- results can be cleanly transformed into the LS classification and vice-versa without confusing the numbers.

As a result, from this point on, I will be using the terms Perfective, Adaptive, Corrective, Preventative in reference to this classification system.

2.3.3 Problem Areas in Maintenance

The third, and rather more under-reported result of the the original surveys, is what the respondents perceived to be the biggest problems in managing software maintenance. The LST survey asked respondents to rate 24 problem areas on a scale of one to five; the LS and NP surveys removed one of those categories while adding a further three, creating a total of 26 problem areas. The top ten responses for each category are presented in table 2.5. In the LST study half of the problem areas are considered technical problems, for example "Adequacy of system design specification", while the other half are non-technical, management issues, for example "Meeting schedule commitments". In the LS and NP studies 12 of the areas are technical while 14 are non-technical, management issues. In table 2.5 the asterisked problem areas are technical issues. As can be seen, in the LST study, seven out of the top ten problem areas were non-technical in nature. In the LS and NP studies, eight out of the top ten problem areas were non-technical. Of most interest is that there is remarkable uniformity between the three studies - the top three problems are the same in each survey - however, the addition of the "Insufficient User Training" category to the LS and NP studies does affect the relative rankings slightly. Table 2.6 contains the rankings without the new category, showing that there is startling similarity between the responses for all three surveys, with responses over the years only changing by a few tenths or hundredths of a point. In tables 2.5 and 2.6 the highlighted cells are those cells that deviate by three or more ranks from the previous study.

The number one perceived problem from all three surveys (with a score of 3.42, 3.20, and 3.29 respectively) was User Demands for Enhancements. However, when performing

Problem	LST	LS	NP
Demand for Enhancements	3.42(1)	3.202(1)	3.289(1)
Documentation Quality*	2.99(2)	3.000(3)	3.173(2)
Competing Demands for Programmer Time	2.95(3)	3.034(2)	3.173(=2)
Original Program Quality*	2.94(4)	2.590(7)	2.577(10)
Meeting Schedule Commitments	2.79(5)	2.686(5)	2.647(7)
Lack of User Understanding of System	2.66(6)	2.608(6)	2.615(8)
Lack of Personnel	2.66(7)	2.576(8)	2.654(6)
Adequacy of System Specification*	2.52(8)	2.428(12)	2.769(5)
Maintenance Personnel Turnover	2.46(9)	2.233(14)	2.412(13)
Unrealistic User Expectations	2.45(10)	2.552(10)	2.808(4)
Program Processing Time Requirements*	2.31(11)	2.554(9)	2.423(12)
Inadequate User Training	NA	2.762(4)	2.596(9)

Table 2.5: Rating of Top Problem Areas

Problem	LST	LS	NP
Demand for Enhancements	3.42(1)	3.202(1)	3.289(1)
Documentation Quality*	2.99(2)	3.000(3)	3.173(2)
Competing Demands for Programmer Time	2.95(3)	3.034(2)	3.173(=2)
Original Program Quality*	2.94(4)	2.590(6)	2.577(9)
Meeting Schedule Commitments	2.79(5)	2.686(4)	2.647(7)
Lack of User Understanding of System	2.66(5)	2.608(5)	2.615(8)
Lack of Personnel	2.66(7)	2.576(7)	2.654(6)
Adequacy of System Specification*	2.52(8)	2.428(11)	2.769(5)
Maintenance Personnel Turnover	2.46(9)	2.233(13)	2.412(13)
Unrealistic User Expectations	2.45(10)	2.552(9)	2.808(4)
Program Processing Time Requirements*	2.31(11)	2.554(8)	2.423(12)

Table 2.6: Rating of Top Problem Areas Without *Inadequate User Training*

factor analysis[3] [28], both LS and NP found that this was not a component of any of the major problem factor groups (i.e. that it wasn't correlated with any other variables, not that it was just missing). One of the factor groups, User Knowledge, consisted of the problems 'Lack of User Understanding of System'; 'Unrealistic User Expectations'; 'Inadequate User Training'; 'Lack of User Interest in the System'; and 'Management Support'. This factor group encompasses all of the problems related to users *apart from* User Requests for Enhancements. Nor did User Demands for Enhancements feature in any of the other factor groups. This means that User Demands for Enhancement was not linked to any measure of maintenance, an increase in it did not result in, say, an increase (or decrease) in the amount of Perfective maintenance performed or an increase in the amount of maintenance done on the system in total. This is particularly interesting for two reasons. The first reason, is a corollary to, Glass [27] and Dekleva's [19] work, which portrays maintenance, and particularly User Enhancement, as a solution rather than a problem; it is seen as a sign of success rather than failure. This view is also shared by the AN study. The argument is that changes to a system are good because they show that it is being used, and that it is considered useful enough that people want it to be altered to cope with new problems. Conversely, it is a bad sign when users want a whole new system when a current system could theoretically be extended to cope with the work. The second reason the result is interesting is that this could also reflect the split of work between unavoidable and voluntary maintenance mentioned in section 2.3.2. In that, the amount of time spent on providing User Enhancements is primarily dictated not by how much of a demand there is for the enhancements but more pragmatically by how much time is available once the unavoidable work is done. As a result this is why User Demands

[3]factor analysis is the technique of trying to identify common groupings of variables whose increase and decrease seem to be correlated

for Enhancement is considered such a problem, managers are tightly constrained by their ability to respond to user requests and as a result it is more their failure to fulfill demands than the demands themselves that is considered problematic.

Something that must be borne in mind when examining these problem areas is that these are perceived problems in *managing* maintenance from the *managers'* perspective. It is notable that "Management Support of System" is one of the lowest ranked problems (at 23rd, 24th, and 22nd respectively). The respondents evidently did not feel that they were a particular problem. Managers' observance of technical problems might very well be through the filter of their programmers, and as a result, problems that are challenging for the programmer might not be challenging for managing maintenance work.

2.3.4 The Most Difficult Work

Although the various studies cited are of a high level view, there are some People oriented facts that can be learnt. Both the MV and AN studies analyse the "difficulty" of different types of maintenance: the MV study by getting programmers to rate the difficulty of a selection of changes, and the AN study by comparing the ratio of total change requests to the total number of days taken to complete the requests, sorted into various categories. The MV study showed that programmers thought that Corrective maintenance was the most difficult type of maintenance to perform with Adaptive (which consists of the Swanson Adaptive category and User Enhancements) as the easiest category. Ratios of work requests to work effort computed from the AN study produce ratios of 1.28 for Corrective Maintenance, 1.27 for Perfective maintenance and 1.11 for Adaptive maintenance. The AN study's ordering of the difficulty of Corrective, Perfective, and Adaptive tasks is the same as in the MV study. Although, as stated, the categories from the two studies are not directly comparable, the dominant component of both Adaptive categories is User Enhancement and the Corrective maintenance category is broadly similar. Combined with the knowledge that providing User Enhancement is not a contributing factor for problems in managing software maintenance, this allows it to said that Enhancing software is one of the easiest tasks maintenance programmers can perform whilst Corrective maintenance is the most difficult. This is a view also shared by Graves and Mockus [29] who thought that Corrective maintenance required 1.8 times more effort than adding code.

2.3.5 Conclusion

Measurement of the goal–oriented results of maintenance programming is not as easy or clear cut as is commonly thought. Confusing and mixed use of terminology, combined with flawed interpretation of the original data, results in a concealment of the extent of the changes in work distribution. Furthermore, these goal oriented studies do not encapsulate the actual work performed by maintenance programmers, but instead show their end results, meaning that the findings are difficult to use to aid or judge maintenance programmers' day-to-day activity.

The biggest perceived problems of managing software maintenance are of a non-technical nature, suggesting non-technical solutions. However, the biggest *perceived* problem, of

Requests for User Enhancements, is arguably not even a problem at all. Furthermore, these perceived problems have not changed over the years, suggesting either that software maintenance research is not reaching practitioners, or it is solving the wrong problems. However, what is clear from the literature is that maintenance consumes the largest proportion of time (and thus resources) of an IT department, whether that proportion is 40 or 90 percent of the department's time, and that that figure has been rising. This means, therefore, that even 30 years after discovering the importance of Software Maintenance, it is still being relatively ignored by researchers, and is the most relatively under-researched field in Software Engineering.

2.4 Lehman's Laws

Lehman's Laws [43] are a series of properties that have been observed to hold true in multiple real-world cases for the development and maintenance of large E-type software systems. E-type systems [42] are defined as systems that are part of the world that they are modelling, and that model themselves and the effects of at least some of their actions. This differentiates them from S- and P-type systems which are systems that model an abstract, formal domain to varying levels of correctness.

There were originally five laws, formulated mostly from the observation of the development of OS/360, but over time three more laws have been added. Table 2.7 provides a description of each law. The Laws could be said to be based on two key ideas: that E-type systems must continually change to remain useful, and that the amount of change will be of a similar amount during each time unit.

It is worth noting that many of these Laws are stated with the provision that they will occur *when there is no intervention*. That is, the effects of these laws can be counteracted if the organisation or team maintaining a system take an active effort to address the issue. For example, law IV was broken by Orbix [55] because they took an active and directed effort to change their maintenance process in a radical way, which, as a result, produced a new dynamic.

Laws I and VI are realisations of the need to for E-type systems to continually change and grow to accommodate user and real-world demands. Laws II and VII are the results of I and IV, due to the continued changed the system loses its original structure and becomes more complex and harder to work with. Laws III, IV, V and VIII deal with the concept that there is only so much work that can be done, as work is performed in a saturated environment. Software maintainers have a backlog of work requests so even working at a theoretical maximum capacity would not result in any more work being done or work being performed to a better standard.

These Laws, backed as they are by decades of empirical research, generally tend to hold true, but breaking down systems into sub-systems can reveal interesting anomalies. Some library sub-systems become so crucial that they develop a massive inertia to change. In my

Law	Description	
I	Continuing Change	A system used in a real-world environment must adapt to the environment or become progressively less useful
II	Increasing Complexity	As a system changes its structure becomes more complex
III	Self-Regulation	System attributes such as change in size and time between releases are invariant
IV	Constant Work Rate	The average effective global effort on a system will remain constant
V	Conservation of Familiarity	On average, the incremental growth tends to remain constant or to decline
VI	Continued Growth	The functionality offered by the system must increase or it will become progressively less useful
VII	Declining Quality	As a system grows and changes the quality of the system will decrease
VIII	Feedback System	The evolution process constitutes multi-level, multi-agent feedback systems and must be treated as such to achieve significant improvement

Table 2.7: Lehman's Laws

interviews with maintenance programmers discussed in chapter 3, I discovered repeated examples of sub-systems that were considered *too important* to risk touching. These sub-systems did not share any commonalities between them except that they provided some core functionality to the rest of the system. In addition, as SeeSoft [22] shows, systems have hotspots: areas of code that attract most of the change. System averages are therefore inapplicable for determining what the typical workload should be for any one particular area.

Another area of interest is in work estimation. The Laws fundamentally state that the amount of work done between each release is constant or declining. This is true when viewing the average work between releases. However, when you examine charts of day-to-day work [11] there are peaks and troughs, often of quite a severe nature, that occur. So, while the average amount of work for the year might suggest that two maintainers are needed, it might be that the actual day-to-day workload could generally be handled by one, except on occasion when four maintainers might be necessary to deal with immediate, urgent workload.

These laws suggest a rather pessimistic view of software, as, taken at face value, they suggest that software will become bloated, complicated and unmanageable. However, as stated above, all of these laws are stated with the assumption of there being no intervention, so a system *can* be made smaller if time is dedicated to performing that task. Orbix reduced its staff but increased the amount of work performed by radically restructuring their maintenance process using Agile methodologies. Refactoring [24] can radically alter the structure of a system, both reducing its size and complexity while increasing its quality.

2.5 Structured Literature Review

Given the absence of any paper demonstrating the dominance of Process or Product issues over People, one would expect to find that research papers were equally split between the three fields, so that one third of the mainstream Software Maintenance research would be about People. However, given general difficulty of finding quality papers on People in Software Maintenance, I decided to examine this issue in more depth. As a result, I undertook a comprehensive review of the literature.

2.5.1 Questions Asked

The exhaustive literature review was undertaken to answer two research trend questions. The two hypothesised research questions are laid out below:

- Hypothesis One: There is sufficient empirical work in the field of software maintenance.
- Hypothesis Two: There is an approximately equal proportion of of papers on People in software maintenance as there is on Products and Processes.

Why Hypothesis One was Tested

Tichy et al. [76] and Zelkowitz and Wallace [86] have both performed research trend reviews of the literature, respectively in Computing Science and Software Engineering. Both papers are of the view that the amount of current empirical work is insufficient when compared to other scientific fields. Furthermore, they find no compelling argument for the lack of empirical work in Computing Science and Software Engineering. As a result it is valuable to see if Software Maintenance has a higher or lower proportion of empirical work. If Software Maintenance has a lower or equal proportion of empirical work then that would suggest that the most useful direction for my future research would be empirically based itself.

Why Hypothesis Two was Tested

Due to the nature of Software Maintenance (being split into People, Processes and Products), it is important that there is an equitable split of research effort on each of the three aspects. This is especially true given the lack of any research definitively showing that one aspect is more important than the others. The review focuses specifically on the People aspect rather than Products or Processes, as this is the area that is suspected to show a deficiency in research. If, as suspected, there is a lack of research focused on People, then this would strongly suggest that any further work I undertake should be People focused.

2.5.2 Review Construction

Selection of Sources

A number of conferences and journals were selected that were considered to represent *mainstream software maintenance* research. I then read every full length paper in these

sources and classified them by four categories (Any Empirical, Empirically Based, Software Maintainers, None) which are described below. Annual statistics, both quantitative and relative, were then computed for each source and as an overall figure. This review was then used as an instrument to answer the two hypotheses presented above.

Three sources were selected as reflecting mainstream software maintenance research:

- International Conference on Software Maintenance (ICSM) [87]
- European Conference on Software Maintenance (CSRM) [88]
- Journal of Software Maintenance and Evolution: Research and Practise (JSMERP) [89]

The JSMERP and ICSM are the two premier publications for maintenance research and it would be impossible to argue that they are not a primary resource for mainstream Software Maintenance research. CSRM is large but slightly less prestigious then ICSM, thus allowing a slightly lower quality of paper in which expands the review from simply the cream of the crop without going all the way to the fringe of the subject. From these sources I have selected only full length papers, that is, those papers of seven pages or longer. I chose to do this as the short papers tended to be more speculative and lacking in substance, and including them would have skewed the statistics. In my estimation, inclusion of short papers would have reduced the ratio of empirical papers in the conferences. This paper size restriction was ignored for the 1988 and 1985 ICSM conferences, as these papers were mostly no longer than six pages, and as such counting only full length papers would have excluded the majority of research presented at those conferences.

Each source was read from its founding date (ICSM: 1985; CSRM: 1997; JSMERP: 1989) to the start date of my Ph.D term, 2001.

Methodology

In a standard systematic literature review, often only the abstract needs to be read to determine if a paper should be included or excluded. In the case of this review it was found that the abstract was often far from sufficient to determine if any empirical work had been performed. This was a problem that Sjøberg et al. [66] had discovered in their review of controlled Software Engineering experiments, where they stated that confusing terminology resulted in many false positive identifications of papers. Kitchenham [37] states that titles and abstracts are exceedingly important when performing systematic literature reviews, however, in Software Engineering titles and abstracts are often insufficient for researchers to rely on, and the content of the paper must be read [8], which vastly increases the length of time required for the review. In my particular case each paper had to be read to establish its empirical content, although often the papers just needed to be skim-read, in search of particular keywords (such as *evaluation, case study, experiment, test*) in section headings or the main text. Papers that met this minimum requirement where then reread in more detail to determine the amount and type of empirical content. This problem was also a factor in determining if an empirical paper discussed actual

software maintainers. In the case of trying to determine if a paper fitted the Maintainer Focused category (described below), a relaxed viewpoint was used: if there was doubt as to whether it should be included or excluded it was included. As this category was being used to determine if there was sufficient research in the the area, it was felt that aggressively discarding papers might bias the result in a negative direction. With the inclusive view, the final figure produced can be considered the *largest possible* amount of work on People, so if it turned out to be below the one third boundary, it could be confidently stated as being insufficient.

Classifications

Here I define the four categories that I used to categorise the papers from the three sources. The categories are: Any Empirical; Empirically Based; Maintainer Focused; None. I have classified each paper in the four categories using true/false.

Any Empirical

A paper would fall into the Any Empirical category if it has any measurements of a real maintenance environment, or any empirical validation of the work it is presenting. It would only be classified as such if the real world example is sufficiently realistic, unless the paper states that the artefact under consideration is designed only for small situations. Contrived programming examples are not counted, nor are toy examples (for example the Sun Pet Store [46] or the oft referenced Gas Station [31]). Therefore, Processes which are demonstrated on 50 line programs when their purported target is 50,000 line programs do not count as containing any empirical work. Self-referential papers, for example, papers reporting on a software tool that is being used to maintain the software tool, or a process improvement methodology that is used to improve the process improvement methodology, could be counted as Any Empirical. As it stands, I have classified those papers as Any Empirical as long as they pass the size threshold. Formal controlled experiments of any size are included in this category as they are covered by the Empirically Based definition, the results of which are also included in this category (see below).

Empirically Based

For a paper to be Empirically Based, the primary purpose of the paper is to measure. A significant portion of the paper is reporting on or analysing data gathered from experimental or real-world maintenance situations. A tool paper that spends the overwhelming majority of the paper presenting statistics or experience reports about the tool being used by a real maintenance team, or experimental results comparing the tool with another approach, would be classified as Empirically Based. Any paper that is Empirically Based is also counted in the Any Empirical category.

Maintainer Focused

This category is a direct mapping for analysing the proportion of papers on People in

Software Maintenance. For a paper to be Maintainer Focused, a significant amount of the paper should discuss issues related to maintainers and maintenance managers. Fundamentally, the paper should have some discussion, report or analysis of human issues in Software Maintenance. This includes papers that primarily discuss other issues but apportion some time to discussing actual maintainers and maintenance managers. As with Any Empirical, there is a judgement call to be made: a paper about a new Process that mentions in a couple of sentences how maintainers reacted to the implementation of the process would not be judged to be Maintainer Focused. On the other hand, a Process paper that discusses human factors in their own right would be classified as Maintainer Focused, even though the bulk of the paper is discussing other issues.

None

This category is used when the paper does not fit any of the above categories.

2.5.3 Basic Discoveries

The classified results of the review for the three sources are presented in tables 2.8-2.10, with table 2.11 containing the combined totals. The highlighted values are those that are greater than the overall average for each category.

Differences and Similarities Between Sources

A Chi-Square test shows no significant difference in the number of Any Empirical or Empirically Based papers between the three sources. However CSRM does have a statistically significant smaller ratio of Maintainer Focused papers. Given the CSRM has a larger number of papers from smaller research groups this could indicate that performing research that involves maintainers requires a relatively large amount of resources.

Trends

Examining ICSM, there is a general increase over the years in the number of papers containing Any Empirical work as seen in table 2.8 and figures 2.1, the last eight years containing above average amounts of Any Empirical work, rising to about 70% of all work from about 50% in the first eight years. There is a similar, but slightly weaker trend with the number of Empirically Based papers, with the number of papers having increased from a particularly low point in the late 1980s / early 1990s. Conversely, there is no discernible change in the number of papers about People, with the number of papers classified as Maintainer Focused staying fairly constant over the years. There are similar trends in JSMERP (table 2.10 and figure 2.3), with an increase in the number of papers both containing some and containing primarily empirical work. From the perspective of empirical research this is encouraging. This also matches the results of Zelkowitz and Wallace who noted an increase in the number of papers containing empirical work over the ten year time span that they examined.

The increasing volume of Any Empirical work must be balanced against the type of work

that is producing that rise. If the rise is purely down to an increase in the number of Empirically Based papers then it is not as encouraging as it could be. Although Empirically Based papers are important, it is also important that advocative work, that is, research proposing new Processes to follow or implement, provides empirical validation of the work otherwise its value is unclear.

Thus it would be encouraging to show that the number of empirically validated papers was growing as well as Empirically Based papers. Table 2.12 shows the percentage of papers for each year that contained Any Empirical work but were **not** Empirically Based. Obviously there are two reasons as to why the percentages could be high or low in the table: the value could be high due to there being very few Empirically Based papers that year, which would be a disappointing result; or it could be high due to there being a large number of empirically validated papers which would be an encouraging result. To aid analysis, those years with an above average amount of Empirically Based work have been highlighted. Highlighted cells which have a large percentage are therefore the most encouraging results for the amount of empirical validation occurring in software maintenance. ICSM, the major source of papers, certainly shows an increase in the amount of Any Empirical work and this increase still exists even when taking into account the increase in Empirically Based papers.

Subjective Commentary

A lot of the papers that were classified as having Any Empirical work had poor levels of empiricism. They mostly involved a simple demonstration of a process or tool to show that it "worked", with very little detail and often without even the most cursory of comparison or analysis with other existing processes or tools that were in the same area.

As the year of publication approached 2000, there was a natural increase in the number of papers mentioning the year 2000 problem, and an increase in the number of papers reporting on the experience of what is termed "massive maintenance". Massive maintenance projects are ones that involve large scale, yet relatively unexpected, one-off need for change, often on systems that have a very low annual change traffic. The year 2000 problem is a classic example of this. Massive maintenance is a slightly anomalous problem area as it does not represent the typical day-to-day challenges that maintenance programmers face. However, the general rise of Any Empirical papers is not solely down to the increase in massive maintenance papers, and their numbers are not so large as to have a disproportionate affect on the results.

2.5.4 Hypothesis One: Comparison With Other Studies

In this section two well known studies of empirical work in software engineering and computing science literature are compared with my own exhaustive review. The two studies selected are Zelkowitz and Wallace [86] and Tichy et al. [76]. Their results are interpreted and presented, in comparison with my own, in table 2.13. There is a third well known study, by Glass et al. into research trends in Software Engineering, however, the level of detail used by Glass et al. makes it impossible to pick out the empirical nature

Year	Total Papers	Any Empirical	Empirically Based	Maintainer Focused
1985	28	17(61%)	10(36%)	5(18%)
1987	20	10(50%)	8(40%)	1(5%)
1988	53	20(38%)	13(25%)	3(6%)
1989	29	15(52%)	8(28%)	2(7%)
1990	25	12(48%)	7(28%)	3(12%)
1991	23	8(35%)	4(17%)	1(4%)
1992	32	14(44%)	8(25%)	3(9%)
1993	35	20(58%)	13(39%)	1(6%)
1994	42	29(69%)	13(31%)	4(10%)
1995	37	26(70%)	15(41%)	3(8%)
1996	34	22(65%)	13(38%)	2(6%)
1997	34	26(76%)	15(44%)	1(3%)
1998	36	29(81%)	18(50%)	4(11%)
1999	49	33(67%)	14(29%)	4(8%)
2000	25	19(76%)	9(36%)	1(4%)
2001	65	43(66%)	20(31%)	9(14%)
Total	568	344(61%)	189(33%)	48(8%)

Table 2.8: ICSM Data

Year	Total Papers	Any Empirical	Empirically Based	Maintainer Focused
1997	17	10(59%)	7(41%)	1(6%)
1998	24	15(63%)	4(17%)	0(%)
1999	18	11(61%)	3(17%)	0(%)
2000	23	18(78%)	9(39%)	1(4%)
2001	19	16(84%)	10(53%)	2(11%)
Total	101	70(69%)	33(33%)	4(4%)

Table 2.9: CSRM Data

Year	Total Papers	Any Empirical	Empirically Based	Maintainer Focused
1989	8	4(50%)	1(13%)	1(13%)
1990	14	6(43%)	4(29%)	2(14%)
1991	11	6(55%)	5(45%)	0(0%)
1992	13	2(15%)	2(15%)	1(8%)
1993	12	6(50%)	3(25%)	0(0%)
1994	16	4(25%)	2(13%)	1(6%)
1995	21	13(62%)	7(33%)	3(14%)
1996	19	13(69%)	8(42%)	1(5%)
1997	17	13(76%)	7(41%)	4(24%)
1998	18	13(72%)	10(56%)	5(28%)
1999	18	15(83%)	7(39%)	1(6%)
2000	16	10(63%)	8(50%)	0(0%)
2001	19	15(79%)	9(47%)	3(16%)
Total	202	120(59%)	73(36%)	22(11%)

Table 2.10: JSMERP Data

Total Papers	Any Empirical	Empirically Based	Maintainer Focused
871	534(61%)	295(34%)	74(8%)

Table 2.11: Overall Data

Year	ICSM	CSRM	JSMERP
1985	25	×	×
1987	10	×	×
1988	13	×	×
1989	24	×	37
1990	20	×	14
1991	18	×	10
1992	19	×	0
1993	19	×	25
1994	38	×	12
1995	29	×	29
1996	27	×	27
1997	32	18	35
1998	31	46	16
1999	38	44	44
2000	40	39	13
2001	35	31	32

Table 2.12: Percentage Any Empirical Work Minus Empirically Based Work

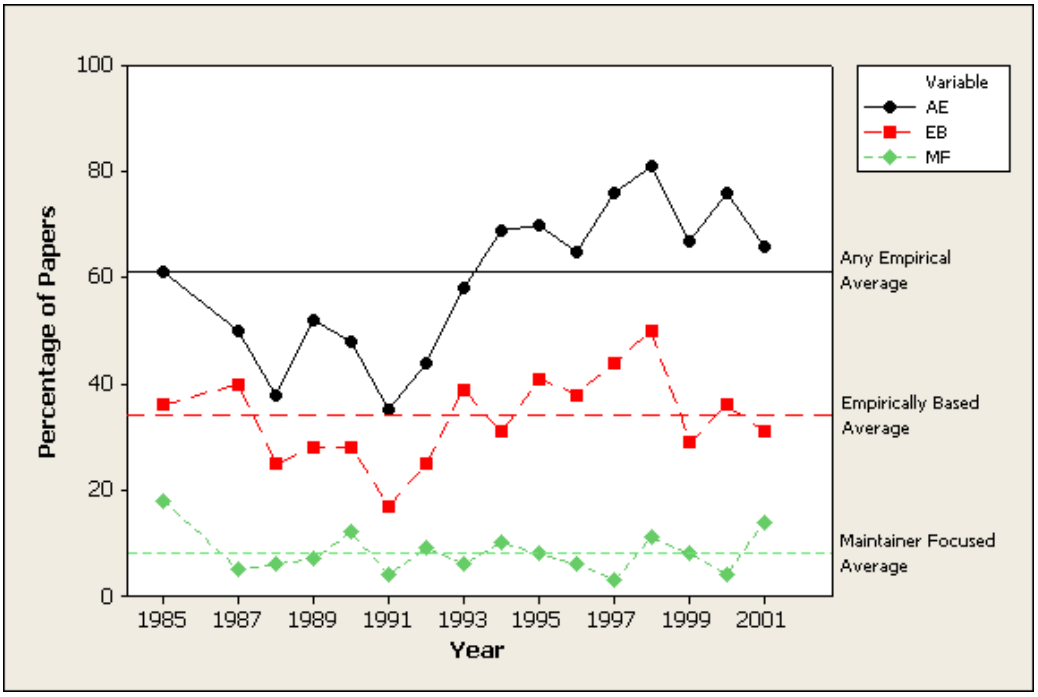

Figure 2.1: ICSM: Percentages of Papers in each Category

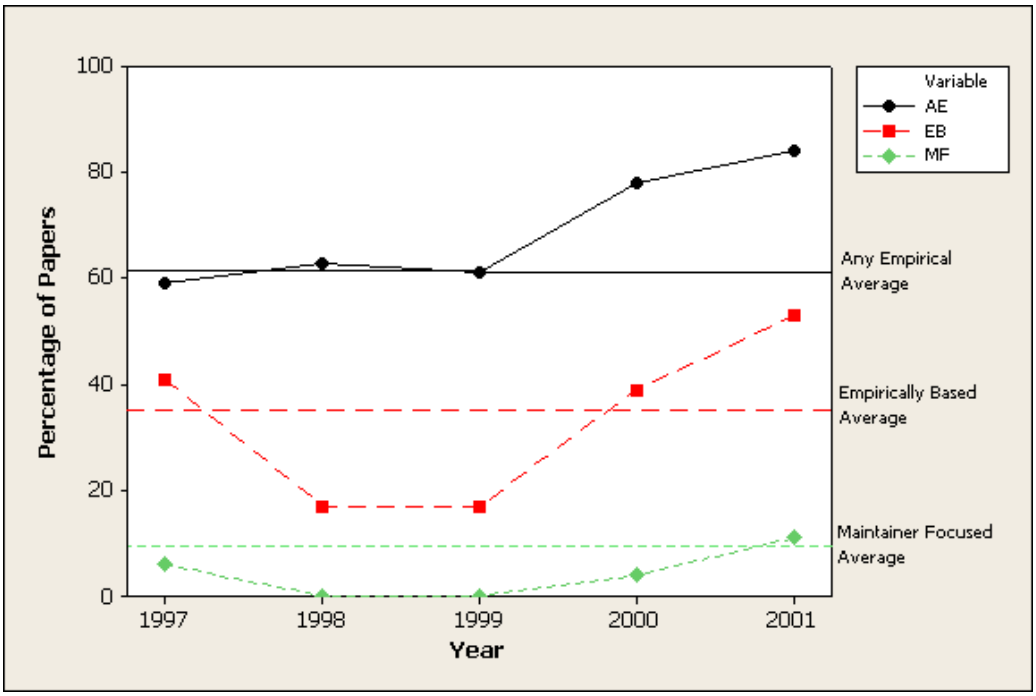

Figure 2.2: CSRM: Percentages of Papers in each Category

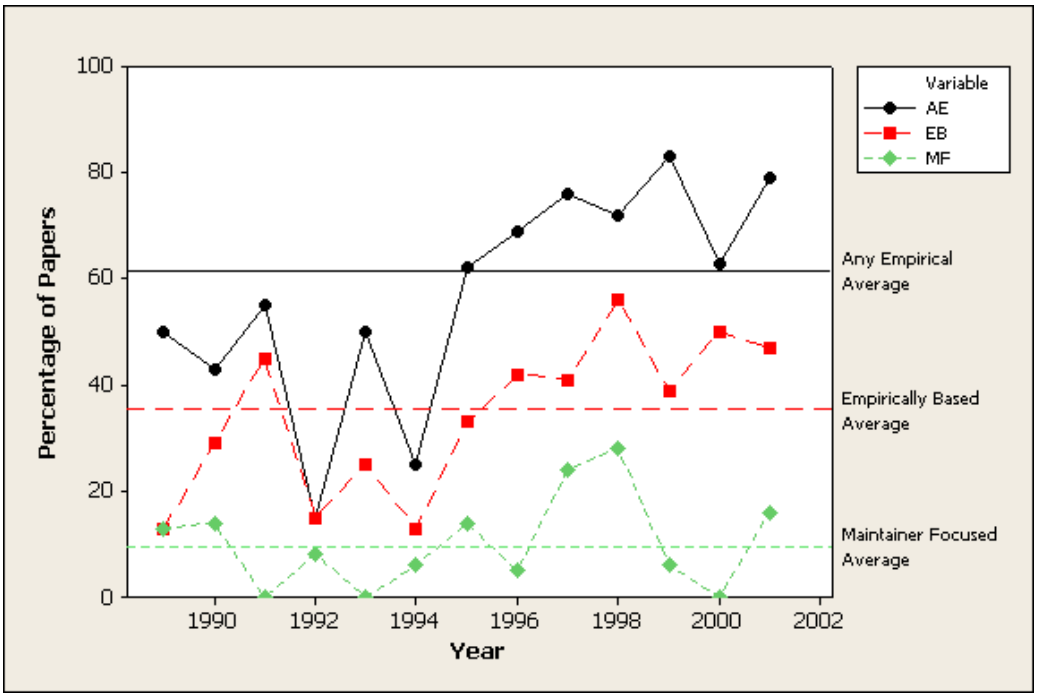

Figure 2.3: JSMERP: Percentages of Papers in each Category

of the papers examined. While the conclusions of Glass et al. are discussed, I do not examine the details of the paper.

Overview of Studies

Zelkowitz and Wallace reviewed 612 papers, although that included book reviews and conference reports, articles that I excluded without counting. Tichy et al. reviewed 256 computing papers plus another 147 non-computing papers. My study examined 871 papers. Both Zelkowitz and Wallace and Tichy et al. selected publications that they thought represented mainstream, general research in their chosen area; the exact publications are detailed below. Zelkowitz and Wallace selected three years, 1985, 1990 and 1995, and then read all papers from the selected publications. Tichy et al. chose to select "recent" publications, which resulted in them selecting papers for some journals from 1991 to 1993, while others only had papers selected from 1993. Once the year range for a journal/conference was selected all papers were examined. In comparison, my study selected all full length papers from the founding date of the selected publications to 2001.

Both Zelkowitz and Wallace and Tichy et al. examined the papers in more detail than my own study. They looked at the exact form of empirical validation, if any, that the paper used. In comparison, I was examining if the paper contained *any* empirical work and examined it no closer. This allowed my study to examine more papers but gather less information about them. Furthermore, my study covers a greater contiguous block of time than either Zelkowitz and Wallace or Tichy et al. Whilst the selection of '85, 90, 95 is probably going to give a reasonable indication of research trends over time, it can be seen that if certain years from ICSM containing the largest peaks and troughs had been selected then a skewed view of the change in research could have been produced.

Both the Zelkowitz and Wallace and Tichy et al. studies were performed by teams. This provided two benefits as compared to my own study. The first is that the additional manpower allowed them to go into more detail: rather than just determining the basic level of empiricism, they determined what type of empirical approach was being used. The second advantage is that multiple readers allow the classifications to be cross-checked. This can help eliminate the risk of systematic classification errors as well as catching random transcription errors.

The Zelkowitz and Wallace Review of Software Engineering

For each of the three selected years (1985, 1990 and 1995), Zelkowitz and Wallace read all the papers in IEEE Transactions of Software Engineering, IEEE Software, and the International Conference of Software Engineering. The papers were classified as to what type of empirical method was used to validate the work presented. One category of particular interest (and difficulty for analysis) is the Assertion classification. This was the single largest category, with 192 of the 612 analysed papers categorised thusly. The category is defined with explicitly damning language:

> "There are many examples of developers being both experimenters and subjects of study. Sometimes this happens during a preliminary test before a

> more formal validation of the technological effectiveness. But all too often the experiment is a weak example favouring the proposed technology over alternatives. As skeptical scientists, we would have to view these experiments as potentially biased since the goal is not to understand the difference between two treatments, but to show that one particular treatment (the newly developed technology) is superior. We call such experiments assertions."

As can be seen, they encountered the same difficulty in determining "true" empiricism as was encountered in my own study. Their solution was to ghettoise the research into its own category, which does impede directly comparing my results to theirs. The Assertion category contains a mixture of papers that I would have classified as Any Empirical, Empirically Based or None. For the purposes of a rough comparison, I will categorise Zelkowitz and Wallace Assertion figures as Any Empirical. This will result in the Any Empirical figure being larger than it should be, however the true value cannot be known and further analysis will be performed bearing in mind that the Any Empirical figure is larger than it should be.

Zelkowitz and Wallace have two categories which would definitely be classified as not meeting the criteria of Any Empirical: Not Applicable and No Experimentation. Together they constitute 35% of the papers Zelkowitz and Wallace examined, although their Not Applicable category includes papers that I excluded from my literature review, such as book reviews, or conference reports. If the Not Applicable category is excluded, the No Experimentation figure would constitute 30% of the remaining papers. The remaining papers of the Zelkowitz and Wallace survey 33% of the total, are in categories that count as Empirically Based. Combining the Assertion category with the Empirically Based categories means that 70% of the papers are counted as Any Empirical.

Tichy et al. review of Computing Science

Tichy et al.'s "broad set" of computing science publications consisted of: ACM Transactions of Computer Systems 1991-1993; ACM Transactions on Programming Languages and Systems (1992-1993); Transactions of Software Engineering 1993; and SIGPLAN Conference on Programming Language Design and Implementation 1993. They also sampled 74 random papers published by the ACM in 1993, rejecting 24 of them for either not being peer reviewed research papers or because the papers were not available to them. They compared their work to two other fields by analysing Neural Computation 1993 and Optical Engineering 1994 and producing comparative figures. They classify papers into five groups: Formal Theory; Design and Modelling; Empirical Work; Hypothesis Testing and Other. Empirical Work and Hypothesis Testing map directly to my Empirically Based category. Formal Theory consists of papers that present pure theoretical work, that is, work which is entirely within the bounds of formal reasoning and mathematical proofs. These are papers that do not require empirical work to be useful. The Other category, from the point of view of determining their empirical content is ill-defined, and is simply described as papers that do not fit into the previous four categories. However, it can be determined that it does not contain any Empirically Based papers as they would be covered by Empirical Work and Hypothesis Testing, nor did Tichy et al. feel it was

necessary to measure the amount of Empirical work the papers contained in the same way as they did for the Design and Modelling papers. As a result I decided to count the Other category as containing no empirical work. So, the Formal Theory and Other categories map to the None category. Design and Modelling covers all papers that may or may not be counted as Any Empirical. For the Design and Modelling category, each paper was examined to count the amount of empirical work that it contained. Five sub-categories existed: 0%, <10%; <20%; <50% and >50%. The third category introduces problems in comparing their results to my own. Twenty-two percent of the Design and Modelling papers were identified as having between 20 and 50 percent of the paper being empirical work. If this category is considered as only meeting the criteria of Any Empirical then only 13% of papers that Tichy examined were Empirically Based. However, if this category is considered sufficient to meet the Empirically Based criteria then 27% of papers are Empirically based. The true determination is probably somewhere in-between these two extremes as papers with say 21% empirical work and less likely to be empirically based than papers with 49% of their content being empirical work. One final point to consider is that Tichy et al. did not consider a demonstration of the system as empirical work. This fact will result in them classifying less papers as containing empirical work than my own approach which accepted as Any Empirical papers that provided a demonstration on a realistic system.

Result

The key figure to examine when comparing my studies to the two other studies is None. If the *actual* levels of empirical work in papers examined in the three studies is similar then when comparing the figures of myself, Zelkowitz and Wallace and Tichy et al. it would be expected that the Zelkowitz and Wallace figure for None would be lower than my own which in turn would be lower than Tichy et al. This is due to the Zelkowitz and Wallace standards for a paper being Any Empirical being looser than my own, which in turn were looser that Tichy et al. As can be seen from table 2.13 this is exactly the case with, respectively 30%, 39% and 48% of papers being identified as having no sufficient empirical work at all. Conversely, the three studies are much more similar for what is considered Empirically Based work, although once again, Tichy et al. is stricter than both Zelkowitz and Wallace and myself. Comparing the values in table 2.13 shows a very close match between Zelkowitz and Wallace and myself while Tichy et al. is, as expected, is a few percentage points lower. Although if you take the stricter view of Empirically Based the Tichy et al. study is vastly lower at 13%

The conclusion of Zelkowitz and Wallace [86], Tichy et al. [76] and Glass et al. [26] is clear: too many papers are produced without evaluation. Zelkowitz and Wallace feel that in the Software Engineering papers they examined, "validation was generally insufficient". Glass et al. state that "There is a severe decoupling between research in the computing field and the state of the practise of the field". Tichy et al. is even more firm, stating that there is active apathy in producing empirical work:

> "Naturally, they are quickly discouraged, and why bother if experimental work is not rewarded and papers are accepted without it?"

In comparison to non-computing fields, computing science and Software Engineering produce a far lower proportion of validated, empirical work. Given the similarity in figures, it is safe to say that Software Maintenance also produces a far lower proportion of validated, empirical work. Both Zelkowitz and Wallace and Tichy et al. are scathing about the quality of empirical work produced describing it as scant and minimal and mostly lacking comparative analysis with either current research or the state of practice. My own impressions of the research in Software Maintenance agrees with this view.

Study	No Empirical	Any Empirical	Empirically Based
Tichy et al. (1995)	48%	52%	27%(13%)
Zelkowitz and Wallace (1998)	30%	70%	33%
Hutton (2007)	39%	61%	34%

Table 2.13: Study Comparison

Answering Hypothesis One

Given that this systematic review has produced a balance of empirical work similar to that of Zelkowitz and Wallace and Tichy et al., and that they concluded that there was not enough empirical work in their field, Hypothesis One is rejected: there *is not* a sufficient level of empirical work in Software Maintenance.

2.5.5 Hypothesis Two: Balance of Research

The systematic review identified that only 8% of papers are Maintainer Focused, which is far less than one third of all papers. As a result Hypothesis Two is also rejected: there *is not* an equal proportion of papers published on People in Software Maintenance. Even when examining Empirically Based papers alone, the majority of years sees less than one third of all Empirically Based papers being Maintainer Focused. Overall, only 25% of Empirically Based papers are Maintainer Focused. Given, as stated, the lack of any paper that shows the dominance of Process or Product over People when it comes to determining what components of Software Maintenance are important, and indeed the evidence which promotes the importance of People [59, 7, 49], this strongly suggests that my work should be focused on the People aspect of Software Maintenance.

2.5.6 Structured Literature Review – Conclusion

I have performed a systematic review of the mainstream Software Maintenance literature. This review, in comparison with similar reviews in the fields of Software Engineering and Computing Science in general, has highlighted two key points: there is a lack of empirical work and validation in Software Maintenance and there is a lack of work examining the role of People in Software Maintenance.

2.6 Conclusion

Analysing the landmark literature shows that while they present much useful information about the Processes and Products involved in Software Maintenance, there is a lack

of information about the role of People. Given the importance of People in Software Maintenance, this gap in the research is an important issue. Empirical research is the only mechanism available to obtain information about Software Maintainers. However, in the field of Software Maintenance, like Software Engineering, and to a lesser extent Computing Science as a whole there is a lack of empirical research. As a first step, before attacking a particular issue, additions need to be made to the body of evidence that exists about People in Software Maintenance - specifically, the role of programmers in Software Maintenance and how they perceive the maintenance Process rather than a management view.

Chapter 3

Interviews With Maintenance Programmers

3.1 Introduction

This chapter primarily covers a series of interviews undertaken with software maintainers in 2003, 2004 and 2005. The intent was to gather a generalised view of maintenance to compare with what little established literature there was. This chapter covers the format, results and analysis of those interviews as well as comparison with the results of similar studies. Finally, a future direction of research is identified based on the results of the survey and the comparison to similar studies.

3.2 Motivation

Given the identified deficit of empirical papers examining People in Software Maintenance, it was felt that it was important to help address this problem. Lacking in the literature, apart from the work of Singer noted below, were descriptions of what maintenance programmers did on a day-to-day basis. Information about how maintainers behaved (beyond examination of thought process for the development and validation of mental models of program understanding) seems limited to being based on "common knowledge" rather than being based on citable research.

A variety of approaches, as discussed in section 3.3.2, were considered, but it was decided that a general survey of maintenance practitioners based on face-to-face interviews would be carried out. Although general in nature, the questions were somewhat focused towards the information gathering strategies of the maintainers, as this was considered to be the field in which there was the greatest possibility for useful future research.

3.3 Basis

3.3.1 Previous Studies

This survey is based on the work of Singer [64] and Singer et al. [65]. Singer performed interviews across 10 corporate groups. Pairs of programmers were interviewed simultaneously. A basic questionnaire was administered, gathering some quantitative data about the participants. Then an interview was undertaken which asked questions about the work practices of the maintainers. The final section of the Singer interviews, trying to identify a tools wish list, was found to be difficult to administer as the maintainers were unaware of what potential tools could be delivered and as a result this section largely fell by the way side. The work practice section of the interview was loosely structured: if a maintainer identified something that they found particularly interesting the questioning was allowed to drift onto that topic. However a core set of questions was retained and mostly administered. From the work practice questions, four common features were identified: source code is king; documentation is untrustworthy; bug tracking systems contain useful knowledge; and problem reproduction is problem solution.

Singer et al. focused on a single team within a single company to identify specific needs to inform tool design. Their approach used multiple survey techniques to create a picture of typical activity by the maintainers which would aid in the construction of a suitable tool. The following techniques were used: a web questionnaire; longitudinal study by interviews and shadowing of a software immigrant; general work practice survey similar to the earlier Singer study administered to all members of the group; selective shadowing of volunteer members of the group; analysis of company wide tool use statistics; and think-aloud analysis of programmers' actions. These multiple views of programmer activity were cross-referenced to produce a picture of the most common activities undertaken by the maintainers. Searching was determined to be the most common activity, so a tool was developed to aid the types of searches that the maintainers performed.

When performing a survey that attempts to identify trends that the researcher wishes to extrapolate to the programming population as a whole, it is more important to survey programmers across different companies than it is to increase the volume of programmers surveyed. If one company is exceptional and 20 programmers from it are interviewed then the exceptional nature will be repeated 20 times. On the other hand, if 10 programmers from 10 different companies have the same problems, this allows a greater degree of generalisability.

3.3.2 Suitability Of Interview Approach

There are many approaches to finding out what software engineers are doing "in the wild". Three approaches were considered: ethnographic study; questionnaire; and interviews. Ethnography involves the study of subjects principally by the shadowing and recording of information about them as they go about their day-to-day activities. A questionnaire based approach follows the simple approach of compiling a list of questions and sending them out to a target population and then gathering the responses. The interview

based approach involves arranging a series of face to face interviews with subjects, where questions are posed and answers given and discussed by the interviewer and interviewee.

All the approaches have their benefits and drawbacks. In its favour, the ethnographic approach produces a large volume of data and is the truest reflection of what the subjects actually do. Set against this is the knowledge that subjects will be behaving differently knowing that they are being watched and recorded. They are also exceedingly time intensive for the ethnographer, to gather information on what a subject did for eight hours, they must spend eight hours with the subject and then spend *at least* as long again categorising and organising the gathered data before it becomes useful information. Questionnaires allow the surveying of a large number of people with minimal time investment, as it is little more effort to mail 500 copies of a questionnaire than it is to mail 200. Questionnaires, however, have problems with low response rates, and low response rates reduce the generalisability of the results. Furthermore, the greater the complexity of the questions, the less likely it is that questionnaires will be completed. Given the lack of access to the researcher, unclear questions are a particular problem, in that inaccurate answers based on faulty assumptions are worse than no answers at all. In some regards, interviews sit between the deep time investment of the ethnographic approach and the quick and easy nature of questionnaires. They cannot obtain the full range of information available to the ethnographic method (which also includes the use of in-depth interviews to validate the observational data), nor, in relation to questionnaires, can it access the same number of subjects for the amount of time invested. However, the interview approach requires far less time investment than ethnographic methods and it can gather far more detail than questionnaire based methods. Not only that, but in the case of poorly worded questions, discussion between interviewer an interviewee allows resolution of confusion and thus allows useful data to be gathered where in a questionnaire confusion could result in poor and misleading answers.

From the perspective of my thesis, the ethnographic method requires too much time to gather a worthwhile, cross referencable amount of information. Furthermore, as a researcher without reputation, it would be extremely difficult to find maintainers willing to be shadowed and companies willing to let the shadowing occur. The questionnaire approach does not give the depth of answer necessary to gain an informed picture of what maintainers do on a day-to-day basis. Only by using free response questions could the problem of preconceived notions affecting the outcome be avoided, and as the time necessary to complete a questionnaire increases so the response rate decreases and the time to analyse the results increases. Interviews offer a balance between time investment and depth of response. By validating answers against other literature a in-depth picture can be built up.

3.3.3 Question Adaptation

The questions used by Singer et al. in the work practices interview section of their survey formed the basis of the questions used in my interviews, as well as being the foundation of the Singer workplace study.

An initial questionnaire was developed to discover both basic demographic information about the subjects and to act as a gentle lead into the main interview questions which started by getting additional system information. Like the Singer et al. study, programmers were asked to sketch and describe the layout of both the overall system and their particular sub-system. This was done to give the interviewer an general idea about the system. Although this consumed time and did not directly relate to the maintainers work practices it was considered vital for providing a solid base for the remainder of the interviews. With a knowledge of how the system operated the interviewer is able to ask questions that are directly related to the system (for example, "how do you go about fixing faults in component A?", "do you do much work with component B") and avoid lines of questioning that are not directly relevant to the characteristics of the system. The following sections, using references to described system, then try to examine the structure of the current maintenance process and then the nature of information they use to perform maintenance and now they go bout finding that information. The interview closes with looking at things the maintainer finds lacking or problematic in the current environment.

The questions were based round the idea of the interviewer having a minimal amount of knowledge about how maintenance is performed in the real-world. As a result questions were with the assumption that discussion would take place as to exactly what the interviewer was looking for. There is no assumption that there is a maintenance process in place, or that it has any implied form, nor is there any assumption about documentation quality or maintainer behaviour. As with the Singer (and Lethbridge) studies if the train of thought of the interviewees diverged from the question then that line of thinking was followed as the interviews are intended to find out what the programmer themselves find most interesting/difficult about maintenance work. This is balanced against the core element of the interviews, examining ways in which maintainers gather and use information to develop their level-of-understanding about a system to perform work on it.

3.3.4 Interview Structure

I wanted to identify what the maintainers thought was most important, not what I had as preconceived notions. Similarly to the the Singer study I wished to interview at least two programmers at each company to help make comparisons but circumstances prevented this in some cases. Companies, specific mangers and programmers were approached to select themselves/others to take part in the interviews. The questions were e-mailed to the participants ahead of time to allow them to determine what was and was not relevant. The questionnaire section was either filled out pre interview or run through in the opening minute of the interview. After confirming the details of the questionnaire the initial system discussion was worked through and then the main questions were started. Most interviews started tying to get a picture of the official maintenance process and after that had been discussed information gathering strategies were the principle topic of discussion.

The interviewees were also contacted by e-mail after the interviews with follow up questions so that I could develop a fuller understanding of their work practices and thought

processes, and also to confirm what I thought were the key points identified in the interview.

3.4 Results

3.4.1 Brief Descriptions of Companies and Programmers

The following section provides an overview of the different companies, systems and programmers examined.

Company A

A small technology company focused around a single, highly configurable, commercially available product which they maintain and enhance.

- Programmer 1 — The sole maintainer of the user interface component of the system.
- Programmer 2 — Works on the back-end tool support for the system along with one other programmer.

Company B

A very large, multi-national, financial services organisation

- Programmer 3 — The head of a maintenance group primarily working on a key internal financial transaction system.
- Programmer 6 — Works in a semi-independent maintenance group under the management of programmer 3. Works on a variety of systems including the key financial transaction system.

Company C

A large multi-national commercial banking group.

- Programmer 4 — Works on the user-interface sub-system of a web banking system.
- Programmer 5 — Works with programmer 4 on the user-interface sub-system of a web banking system.

Company D

A very large, multi-national, financial services organisation.

- Programmer 7 — Worked in a development and maintenance group primarily focused on a key internal financial transaction system.

University E

A university Computing Science department.

- (Programmer 8) — Not fully interviewed due to not having enough experience with the system.
- Programmer 9 — Working on a short term (One and a half month) contract in a small group upgrading and bug-fixing some research project code.

Company F

A very large international defence contractor.

- Programmer 10 — Maintained a number of small-to-medium sized systems, often embedded software, working with a number of often ad-hoc groups.

Company G

A very large financial information service and brokerage company.

- Programmer 11 — Maintained and developed numerous sub-systems all of which interacted with, and formed part of, the companies main, commercially used, transaction processing system.

3.4.2 Basic Demographics

The questionnaire part of the survey was designed as a gentle lead in, however a small amount of demographic information was collected, which is presented in tables 3.1 to 3.4. Programmer 10 felt unable to answer the question of system age and experience with the system as they maintained a variety of different system rather than being focused on a single one. All but one of the programmers were male. All programmers indicated proficiency with at least two programming languages but most were unsure as to how many they should record as they felt they were 'capable enough' without necessarily having fully mastered a language. As a result I have excluded that figure from the tables. System age was split in a fairly normal distribution between the five age categories, most systems being 3-8 years old. Like the Singer study programmers were unable to give lines-of-code estimates for the size of the systems they worked on, most more able to talk about number of modules or packages. This was in part due to the heterogeneous nature of the systems and also due to the fact that, as section 3.4.4, demonstrates, maintainers specialise on sub-systems and so do not have good level-of-understanding of parts of the system outside their area of knowledge.

This information paints an image of the interviewed programmers as typical programmers working on typical systems.

18-25	26-35	35-45	46-55	55+
2	5	2	1	0

Table 3.1: Age Ranges

<1	1-3	3-8	8+
1	2	4	3

Table 3.2: Years of Professional Experience

3.4.3 Assignment

Maintenance Priority Levels

The various companies and programmers operate under a system of change request importance. In some cases this was formally defined, as with Company G, while in others it is just a intuition of the programmers, as with Company A. The other companies had varying levels of formalisation of work importance. In general, the work can be split into three levels:

1. Enhancement — features that are to be added in the next release. Something that has to be done in the next quarter or half year.

2. Non-severe bug — small problems that can be worked around for now but need to be fixed. Often only being experienced by one user, but could happen to more. Seems to have a time frame of a week to be fixed and then will be rolled out either in the next release or, more rarely, will be sent out in a special patch.

3. Severe bug — One that is causing the system to fundamentally fail. In the case of financial institutions, this could be costing the company millions of pounds per minute or the potential for large lawsuits from clients. These are "drop everything" bugs that require the programmer to concentrate on nothing else. Will be released to the live system as soon as possible and will have testing done on it after the fact.

Company G defined "Severe Bugs" very formally and gave programmers specified powers to allow them to solve the problem. Specifically, programmers were allowed to contact anyone they thought was important and demand their help to work on the problem. The full weight of the company was behind them. I feel that this empowerment of programmers in this crucial situation is vitally important. Without it, the programmers could be left paralysed, unsure of what they can and cannot do. With it, they can make sure action is taken without repercussion. In company D having to get someone in to help with "your" mess might be seen as a sign of weakness, but this type of reaction could be potentially devastating for the well being of their company.

<1	1-3	3-8	8-15	15+
1	2	4	1	1

Table 3.3: Age of System

<1	1-3	3-8	8+
1	6	2	0

Table 3.4: Time Working with System

Programmers at all organisations were also able to self-manage their workload to a fairly significant degree. With the exception of a severe bug hitting their inbox they were fairly interactively involved with their management for deterring priorities within the bounds of testing and release deadlines. Due to the heavy formalism that surrounded the testing and release processes at companies they were the only dates that mattered. As long as the work was ready for these deadlines the companies didn't mind how the work was produced. This meant that there was very little that defined maintenance requests in the companies, maintenance requests would exist, and be formally recorded but up until the point they were completed there was little if any indication of how that work was progressing bar informal discussion with management. This rests hand-in-hand with the lack of coding conventions identified in section 3.4.5.

Poor Assignment Accuracy

There is an implicit, and sometimes not formally recognised, first step in the corrective maintenance process that has been identified by all the programmers. After a bug has been raised, there is the need to identify the area of the system where the bug is being generated and thus the programmer whose responsibility it is to fix it. Programmer 7 stated that the largest problem in their group was the inability to raise a potential problem without also being assigned the task of fixing it. Programmer 1 stated that around 50% of all bugs assigned to him were not his responsibility and were 'thrown back' to the group after initial investigation. He felt that this was because of the nature of his system: being the GUI component, it was what customers experienced the most and so thought of as the bug location when the true cause was deep in the back-end of the system. Programmer 11 estimated that around 33% of bugs assigned to him would be reassigned as the "true" cause was uncovered, with no guarantee that the bug would not be reassigned a second time - and this was with a front line support desk that he rated as "excellent". Programmers 4 & 5 also spend a significant amount of time working out if it is their sub-system or another part of the system as a whole that has the error. The support desk tries to classify the bug and attach as much additional information that they can get. Once again there is a tricky balance: maintenance requests need to be fulfilled quickly, and time that is spent trying to accurately determine who should fix a bug could be being spent actually fixing it. However, some of the maintenance programmers spent a significant amount of time reassigning bugs. If the bug assignment had been more accurate to begin with, multiple programmers' time would have been saved.

All programmers found that the reproduction of errors was a difficult task, and they have all developed a habit of checking to see if their particular sub-system is the source of the error. Programmers 1, 4 & 5 all worked on user interface sub-systems, which, by their nature, tended to attract erroneous bug reports from users. This is due to the fact that

they were the visible portion of the system, so bug reports would often take the form of "the wrong value is being displayed", which initially, at least, would be classed as a UI error. It is only after the programmers have gained access to more detailed information, tested and attempted to recreate the bug, that they can see, for example, that they are being passed bad data, and not that they are presenting it incorrectly. Programmer 11 has the issue of a very large multi-faceted system, of which no one person has a good overview, meaning that a bug will often be passed through multiple programmers while the true cause of the bad data is tracked down.

3.4.4 Discovery

Live System Information

The greatest similarity in the work habits of the maintenance programmers was the use of program logs, and other live information, to aid debugging. Almost all the systems that the programmers worked on produced logs of their execution, and it was those, along with a description of the bug, that the programmers first consulted to help them find out what was going on.

Programmers 1, 2, 3, 6 & 7 all made extensive and expert use of debuggers to debug software. The debugger was often used to "zone" areas of the software under investigation, by which I mean that break & watch points were placed before and after pieces of code suspected of being at fault, so that values could be watched going into and out of the code. If correct values went into the code and incorrect values came out, then it was that zone of code that was at fault. If correct values went in and out or an incorrect value went in to the zone, then obviously another zone was at fault. Once an incorrect zone of code was identified, the programmers would either try to divide the code into sub-zones, iteratively working down to the exact line(s) of error, or by using knowledge of the code place speculative break points at the suspected line(s) of error to try and catch the error immediately. Programmer 7 also used the debugger to examine the final state of a system using the core dump.

With the exception of company F's system, all the systems maintained produced varying levels of logging information. Company A's system produced a very minimal amount of logging, and what logs were there had been inserted on an ad-hoc basis by the maintainers in areas that they had identified as being hot-spots or trouble areas. In companies B, C & D the logs were more thorough, as the systems could potentially deal with large amounts of money, so the system logs were a mandated part of the design of the systems. Transactions were logged, and communication between sub-systems was recorded. Fundamentally, values would be known coming into and out of a sub-system. In company G, logs had an even greater level of detail, in that every action that a user might undertake would be logged, and every mouse click on the screen would be recorded, as well as the same type of logs as in other companies being recorded but at an even greater level of detail. Logs, no matter what level of detail they provided, were used as vital sources of information by all maintainers that had access to them. They allowed the dynamic analysis that only executing the system could provide, but from a static context. They also aided

in the problem of recreating bugs, an activity that is notoriously difficult. By referencing log information, programmers could either negate the need to recreate the exact system setup or get a headstart in how to recreate the conditions that caused the problem. At company G, the detail allowed programmer 11 to ascertain whether the submitted bug report was truthful or not: in one case the user's complaint that "I didn't get notice of an important trade, the system is bugged", was found to be false when, upon checking the logs programmer 11 realised that the user was not even logged into his terminal at the time the trade was offered.

This use of live system information was universal among all programmers. The level of live system information available was variable but it was always used when available. Programmers 4 & 5 just used program logs rather than the debugger. When questioned about the debugger, programmer 4 stated that he "should probably use it more, but it is difficult to set up" due to the nature of their sub-system. So the desire to use live system information is still there, it is just the time investment that is a problem. I think that the value of live system information is clear, but perhaps there is a potential for research into how programmers use that information.

Maintainer's Knowledge

All the programmers were experts on a specific part of the system that they maintained. In Company A, programmer 1 had sole responsibility for the sub-system that he maintained, while programmer 2 was specialised in a part of a sub-system he maintained having a working knowledge of the rest of the sub-system. Programmer 3 from Company B was similar, although due to his position he had a greater understanding of the system as a whole. In Company C, programmers 4 & 5 both covered the same sub-system in its entirety in a similar way to programmer 1. Company A has had a history of programmers specialising in one part of the system which did leave them vulnerable should a programmer have been "struck by lightning". This is being rectified by a policy of cross training to spread system knowledge around the company so that it is no longer held by a single individual. However, this does not reduce the importance of the sub-system specialist's role, as they are still the primary source of work and knowledge about their particular part of the system. At Company B the idea of cross training is already institutionalised and so multiple programmers would have to be lost before parts of the system became unknown. At Company C, both programmers interviewed covered the entirety of the sub-system, so should one be lost then the other would be able to continue working with no significant loss of knowledge about the system. In Company B, programmer 6 was, at the time of interview, a one-person programming team, their partner having just taken a new job leaving him as the sole source of knowledge for the system they maintained. While his partner was still at the company, they practised a policy of reviewing each other's code to make sure neither of them became single points of failure. The problem of Programmer 6 being the sole source of knowledge was somewhat mitigated by the documentation policy of the company, as detailed in section 3.4.6. Programmer 7 from Company D was similar to the programmers from Company A. They all specialised in a single piece of the system, and each sub-system they wrote became their responsibility to maintain. Programmer

10 worked on a number of unrelated systems at Company F. On some he was the sole maintainer, after the system had been written by someone else, while on others he worked as part of a small team of no more than three people. Due to the institutionalised use of code inspections, system knowledge was spread between maintenance programmers. In Computing Department E, programmer 9 only had a small amount of time to work with the system, as his group had made the decision to explicitly specialise in different sub-systems as they would not have the time to learn it all. As a result, programmer 9 became an expert about a particular sub-system in a similar manner to the other programmers interviewed. Programmer 11 from Company G has a different set of circumstances which are detailed below.

In general, the maintenance programmers interviewed specialised in their work. Although company B had institutionalised cross training, that still did not stop programmers becoming individual points of failure, as in the case of programmer 6. Nor did it stop programmers specialising, as programmer 3 still understood parts of the system *much* better than any other programmer on the team, and his loss would significantly impact productivity for code changes in his specialised sub-systems. However, although programmers specialised, each system would also have a programmer or manager who had a good overall knowledge of it. This person would often be an original developer of the system or a particularly long term maintainer. This person could be used by any maintainer to answer questions about parts of the system they did not understand, especially design rationale, without having to track down the specific person who knew exactly what that part of the system was doing. For example, amongst the programmers I interviewed, programmer 3 fulfilled these criteria.

Programmer 11 was in a seemingly different situation from the other programmers. Company G's policies had led to the situation where each programmer had overlapping knowledge of various sub-systems, so that no one programmer had irreplaceable knowledge about any particular part of the system. However, the overall system was so large and multi-faceted that there was no-one with an overview of the system, no-one to go to that could pull it all together. There seems to have been a trade off that has taken place, even if it is not a conscious one. No individual programmer is *technically* irreplaceable as their knowledge is, generally, replicated across at least one other programmer, which is clearly a desirable state of being for the company. However, the lack of an overall view means that architectural changes to the system will be exorbitantly costly as dozens, if not hundreds, of separate programmers will need to be consulted to put together a picture of the overall construction of the system. As a result this ironically creates "vertically integrated" experts. Although several programmers understand how a particular function operates, only one of them knows exactly how it operates in relation to a specific half dozen other functions. This creates a situation where modules contain multiple variations of a single, seemingly well understood function, as every programmer is too frightened to actually change a function for fear of its unknown interactions with other parts of the system.

On large systems programmers specialise on one part of the system. Despite the problem that creates, of risking individual programmers with knowledge, there are good practi-

cal reasons for this. A programmer with specialised knowledge will be able to complete work on a sub-system faster than a programmer with a more general knowledge of the whole system. This approach also reflects the manner in which expertise naturally falls: if a group of programmers were to start working on a system, all with a basic general knowledge of it, as soon as one of them fixes a bug in a particular part of the system, he will have gained a greater level-of-understanding of that sub-system than the other programmers. This then makes him the natural choice to fulfil any other maintenance request that is related to that sub-system, making him more and more specialised as work continues.

Sub-system specialisation is not inherently a problem. However, there are two problems that can be caused by sub-system specialisation, the first being trivial, but the second being of greater concern. The first problem is of over specialisation: theoretically, a programmer could spend so much time concentrating on "their" sub-system that they lose knowledge of how the sub-system fits into the overall system. Without that knowledge the programmer will become less effective as they will have to relearn knowledge about other parts of the system when incorrectly targeted maintenance requests are sent to them. I have not observed this from my interviews and it seems to me to be a theoretical problem only. The second, more pressing problem, is of "sole" experts: if a single programmer is an expert on one part of the system then they become single points of failure. If they become unavailable for any reason, then they take the only source of expertise about the sub-system with them. Company A has suffered exactly this problem: programmer 1 was brought in to replace a programmer who was leaving the company. The plan was for the outgoing programmer to teach programmer 1 about the sub-system of which he was the sole maintainer in a three week period before he left. Instead, the outgoing programmer took his three weeks of vacation time that he was due. Programmer 1 was left with no guidance beyond what the system wide expert could give, and there was no-one who understood the details of the inner workings of the sub-system. Maintenance work on the sub-system effectively came to a halt for the time he took to learn about it.

3.4.5 Implementation

Adding New Code

Almost all programmers stated that they very rarely introduced new bugs into the system when fixing old bugs, although their reasons for this were slightly different. For programmers 1 & 2, they had such a firm grasp of their respective sub-systems that they always knew what they were doing in relation to the rest of the sub-system. As they were the ones responsible for their sub-system, they also know that any bugs they introduce, they will have to fix. For programmers 3, 4 & 5, their low bug creation rate is due to the extensive testing process that happens whenever code changes are introduced to the production system. For programmers 4 & 5 this testing process normally takes up to two months, so the knowledge that a badly written fix to feature X can result in the testing being stopped after a month and a half, and the code coming back with a note saying feature Y is now broken adds extra motivation above professional pride in their work. Programmer 6 uses unit testing and stated that it was the single most useful thing

that he had done. He claimed that it prevented them passing any buggy code onto the testing phase. Programmer 7 stated that the group itself was very competitive, and that individuals are expected to sink or swim by themselves, so there is a very strong desire to prove that you are a competent programmer. This apparently leads to rigorous self discipline. Programmer 10 stated that due to the maintenance process being so rigorous, each change to the system was so small, well tested and inspected that it was very rare for new bugs to be introduced. Programmer 11 stated that new bugs did get introduced when making changes, but that the frequent roll-out and testing cycle the company used meant that they were mostly spotted and fixed before they were redeployed to customers.

This is not to say that the programmers do not write buggy code: far from it. They are human and thus will make mistakes and they spend a large amount of their time hunting down and fixing errors that they have made. However, this result is that they rarely produce buggy code that goes forward to formal testing or is rolled out to customers, which is an entirely different issue. As can be seen from the graphs in Burk and Kung [11] the number of corrective maintenance requests goes down over time, but if the maintainers were continually writing buggy code that made it's way into production then that line would either remain constant or rise.

No Mandated Coding Conventions

Although a company may have a mandated software creation process, what that actually meant for the programmers on a day to day basis was very little, except for programmer 10 who worked in a CMM level 5 environment and so had to produce a large amount of documentation to accompany any work he did. In general programmers did not have to jump though many hoops to produce code: I did not even discover a standard coding style at any of the companies. In company A there has been much debate about a consistent coding style and the pros and cons that it would have, but the decision at the time of the interviews was still in limbo. Similarly there are no mandated tools that the programmers use, and no particular IDE or text editor has been rolled out across the team or company. At company A there are clusters of tool usage where the programmers have found them agreeable and spread their usage around the company. There is a similar situation for companies B, C, D, E & F as well, in that the programmers use similar tools not because of mandating but because they find them useful. In most companies there was not even a mandated language. In programmer 7's overall team (in company D), some developed sub-systems in Java, some in C++ and one group in C. The groups had the authority to choose what they thought was the most appropriate language. At company G things were slightly more constrained, as the central systems were written in Fortran and sub-systems had to be written in C. The e-mail system the company uses is integrated into the change request system, allowing anyone to track the status of change requests as attached to individual programmers. However, programmers still had the choice of using whatever development environment they liked with which to produce code.

3.4.6 Learning

Documentation

The major difference between the programmers was the level of documentation they had about the system. Company A had very little in-code and system level documentation for the programmers to work with. The company had no policy on documentation, so some programmers did document code and retroactively add documentation to previously undocumented parts of the system when they worked, while others did not. On the other hand, the team programmer 3 (company B) worked in had a rigorous documentation policy. An accepted part of the maintenance policy for code is the updating of the documentation, both internal to the source code and external documentation. The external documentation was kept in a Lotus Notes database with all the search and collaborative facilities that that provides. This external documentation consists of design notes, e-mail conversations, and anything that is linked to the piece of code to which the documentation relates. Company G had a similar but even more stringent system of automatically collecting every e-mail relevant to a maintenance request, yet, at the same time, programmer 11 thought that the code level documentation for the system was very poor. For programmers 4 & 5 (company C) there was architectural documentation, but it is so out of date that they would advise newcomers to the system to ignore it. However, there was code level documentation that was kept up to date by themselves and their team leader. Programmer 9 (company E) had no documentation to work with. Part of their job brief was to create documentation for the system, however, time constraints meant that he decided not to do it. Programmers at companies C,D and G all had a similar comment on the state of the documentation. They stated that it was terrible but improving. Small scale, local group driven, Knowledge Bases (they each independently used his term) had been created into which new information was being put. This represented a break from legacy documentation attached to the system and the new information was considered of a much higher quality. However, it could be that this is a short term improvement and over time, as the volume of documentation increases, it starts to fall more and more out of synch as the effort of updating it increases.

One possible reason for the lack of documentation relates to the programmer specialisation. As a maintainer focuses on a particular area they develop a high level-of-understanding about it, as a result they have the *least* need of any member of the team to consult documentation about that part of the system. Even when looking at an area of responsibility that they had not examined in a while none of the interviewees stated that they needed documentation to help them, as beacons and connections to well understood code were all the help they needed for jogging their memory about what the code does. The paradox about the maintainers lack of need is that the maintainer is the person best suited to create that documentation due to their high level-of-understanding. Unless mandated by the company, this forms the crux of causes for documentation to fall by the way side: there is no personal motivation for the maintainer. Even when re-examining code that had not been worked on in a while the maintainers stated that they were able to use references to code about which they had a high level-of-understanding as well as picking out beacons and partial reminders to, relatively, quickly reconstruct rationale for

the code.

Training

Companies had varying approaches to teaching new maintainers about the system they were to maintain. Almost all the companies had some form of introductory course where basic technologies and company ethos were introduced, but when it came to teaching about the software system in particular, approaches were a lot more spotty. As a direct comparison, companies B & D are comparable in business objectives and company size. Furthermore, the systems that the programmers I interviewed from these two companies were working on were broadly similar, being financial transaction servers for internal trades. However, the company training policies were widely divergent. To start working on the system for company B, a programmer has to serve a structured apprenticeship at the head offices for several months under a mentor having to meet several milestones to demonstrate a sufficient level-of-understanding. Before they are finally certified to work on the system, they must demonstrate that they have met the prescribed milestones. For the group examined in company D, the training is practically non-existent: after the initial orientation phase which does not deal with the specific software system at all, new programmers were being asked to deal with change requests from the moment they took their desk. Programmers are expected to get on with it and prove their worth by directing their own learning. Being seen to ask the right kind of questions of the correct people was another way of earning the group's respect, although asking the wrong kind of questions had a large social penalty attached. In companies A & C, it was intended that software immigrants should learn under the wing of an experienced system maintainer, but as has been discussed in 3.4.4, this might not be what happens in practice. The general view of software system training in companies other than company B was as an afterthought: it was considered that software immigrants would be able to ask questions and consult a mentor.

The programmers were asked about how they would train new maintenance programmers freshly assigned to their sub-system. All except programmers 7 & 11 recommended essentially the same method. This was to start by giving the new programmer an overview of the system as a whole: what the sub-systems do, how they communicate with each other, explain the design rationale behind the system and sub-system construction, and who the specialist is for each sub-system. They are then taught about the specific sub-system that they are to maintain, again being given an overview, demonstrating its behaviour in various typical modes of operation, and showing good starting points for debugging. This training method shows a very top down approach to the teaching process. I believe this shows that experienced maintenance programmers place a great deal of emphasis on high level abstractions of the system, or at the very least that they consider them to be the hardest thing to learn and most useful for an immigrant to learn.

3.5 Comparison

Comparing my results with both the Singer general study and the Singer et al. study there are a number of similarities.

3.5.1 Source Code is King

The primary similarity is the use of source code as the prime source of information. The subjects in my study altered that somewhat: they believed that live system information is the prime source of information, as one can misunderstand how an algorithm theoretically operates but there can be no misunderstanding the actual results. If the actual results are different to what is expected, then the flaw is in the programmer's understanding of the implementation of the algorithm rather than the algorithm itself. However, the statement that "source code is king" is concordant with the lack of, and distrust of, external sources of information. They only other source of information that is trusted anywhere near as much as the source code is another maintainer's expert knowledge of the system.

3.5.2 Documentation is not King

Also in line with the findings on my survey is Singer's result of maintainers not trusting the documentation. Similarly to my own results, Singer's companies had varying ways of dealing with documentation, from formal systems to entirely ad-hoc approaches. Another observation was that documentation was useful for a high-level view of the system. This keeps in line with my own findings which suggest that in general, at the high level, systems remain fairly static: a pay-roll system is a pay-roll system no matter how much additional functionality accretes, but as more detail is required the documentation breaks down. The Singer study also echoed the view that programmers see limited value in creating documentation when they are the sole expert on the part of a system they are supposed to create documentation for. That said, the Singer study also showed that bug-tracking databases were used and kept up-to-date. Singer offers the hypothesis that the bug-tracking databases gave a higher perceived value than documentation or that the bug-tracking databases are seen as a form of company wide communication. My interviews offer an alternative view. The bug-tracking databases that were used by companies A, C & G were all integrated with the work assignment system. Bugs within the system were assigned to specific programmers and the tracking system was used by management to view and check progress. As a result it was in the maintainers' best interest to give an accurate view of their level of work to avoid being over-burdened.

The Singer et al. study raises a seemingly anomalous result when looking at the type of work maintainers perform. In the study they surveyed 13 maintainers who all worked on the same system. Six of them responded to an initial web survey where they had to identify what major types of work they did in a free response manner. All six noted that they looked at system documentation, and this was the only work type that was universally identified (although there are some varying levels base assumptions amongst the programmers as not all of them stated that they worked with the source code). Given

the results of both my own and the Singer study, which show that programmers do not trust documentation, this is highly interesting and suggests that these programmers deviated from the norm. As a follow up, eight of the programmer's (including the six who responded to the web questionnaire) were shadowed for an hour each and their actions recorded. Of the 356 total actions recorded only 12 involved looking at documentation. As the Singer et al. study states this seems to suggest that the programmers find checking the documentation so unusual that it sticks in their minds in a way that other activities do not.

3.5.3 Training

The method interviewees suggested for training software immigrants bears comparison with the work of Berlin [6], which examines the mentor immigrant relationship and what kind of information the mentors pass on to immigrants. One of the main pieces of information shown to be passed on was design rationale: explaining why the software was designed the way it was. This matches the type of information the interviewees said that they would pass on to a hypothetical software immigrants.

3.5.4 Programmer Estimation of Work vs Managerial Estimations of Work

In the Singer et al. study, programmers estimated their work split as being 57% bug-fixing and 43% other work. In my study, the programmers' general view was that they spent more time fixing bugs than any other programming task, with only programmers 7&10 estimating that they spent more time on development than bug fixing. As noted, this is not the same as spending the majority of their time fulfilling corrective maintenance requests, this is time spent bug-fixing. These bugs could arise as they undertake perfective maintenance.

3.5.5 Experience

The Singer study maintainers had a high average length of experience with the system they were maintaining,the Singer average being 4.38 years on a single project. This is similar to my maintainers, with the majority being in the 1-3 or 3-8 years of experience with one system. Only programmer 9, who was working on a short term contract of a few months, had less than one year of experience. An estimated average time with system would be around three years.

3.6 Summary

There are a variety of interesting results from this survey, some of which are dealt with in chapter 7. However, four of the results, also seen in the Singer study – relating to system information and how it is gained, presented and stored – combine together to form an interesting picture when related to software immigrants. The four results are:

1. Software Maintainers specialise in the section of the system they maintain.

2. Companies do not have much, if anything, in the way of defined training processes for new maintainers.

3. Exterior sources of information, documentation and mentors, are not always available. Even when documentation is available it is mostly of a very low quality and is not trusted.

4. In the absence of useful exterior sources, maintainers trust live system information above all else.

3.6.1 Result 1: Maintainers Specialise

The picture of typical maintenance this survey portrays is one where maintenance programmers are often the sole programmers in charge of individual parts of the system. Whilst they have a generalised level-of-understanding about how the system as a whole works, they may have no knowledge of the practical implementation of any sub-system besides their own. However, for their own particular sub-systems, they embody a detailed and high level-of-understanding. In many cases this high level-of-understanding does not overlap with the understanding of other maintainers, as, even though multiple maintainers may work on a single sub-system, the natural flow of maintenance requests will result in maintainer having unique knowledge about particular parts of the system. As a result, the loss of a maintainer means the loss of their unique knowledge of the sub-system.

3.6.2 Result 2: No Defined Training Process

With the exception of company B, none of the companies/maintenance groups had a defined training process for letting software immigrants gain a level-of-understanding about the system they would maintain. A software immigrant, after a short 'introduction-to-the-company' training period, would effectively be left to their own devices to gain a level-of-understanding about the system sufficient to perform useful work on it. This is a similar result to the one found in the work of Taylor et al. [74], discussed in greater depth in the following chapter, which also found little in the way of defined training methodologies for software immigrants in business. This lack of training is a clear gap in the current state of practice in software maintenance. Given that software immigrants are often brought in as direct replacements for departing team members, who have unique knowledge, it is surprising that more thought has not been given to passing that knowledge on to software immigrants.

3.6.3 Result 3: Exterior Sources of Information

Documentation is not always available, unless it is a strictly mandated policy of the company (B, F, G), and even when documentation exists there is no guarantee as to its quality. In company G, where every scrap of information about system development and maintenance was saved, the quality of the documentation was rated as being very low. In company F, even given that the company was at CMM level 5, there were still gaps in the documentation that meant programmers had to be personally questioned in order to discover any in-depth information about the system. Sometimes the necessary

programmer had left the company and the design rationale had to be reproduced from scratch by using the code. Combined with result 1, this starts to form a challenging picture for software immigrants. No system training is provided, the documentation that is supposed to describe the system is either missing or inaccurate, and the only other sources of information available, in the maintainer who works on that particular sub-system, could well have been replaced by the software immigrant themselves, leaving no source of information about the code other than the code itself.

3.6.4 Result 4: Programmers Trust Live System Information

This was the strongest result of both my own and the Singer studies: apart from knowledgeable programmers the *only* source of information maintainers consider to be trustworthy is the source code itself. This means that even when documentation is available maintainers do not consider it a trustworthy source of information. Furthermore, as the results above show, programmers with specific knowledge of a sub-system are not always available. This means that software systems are being maintained in environments where there are no sources of trusted, useful information about the program apart from how the program actually operates. This is a very challenging environment for software immigrants to come into as they do not have any current knowledge of how the system operates, nor is there anything or anyone for them to consult. However, from my interviews this does not seem to be a particularly uncommon occurrence, as both programmer 1 and programmer 9 went through exactly this situation.

3.6.5 Discussion

These four results combine together to form a very challenging picture for the software immigrants going into the typical maintenance team in a typical company: a rugged environment of minimal help and support, not because there is a lack of willingness but because there is a lack of resources. The creation of these resources, be they better documentation or training manuals, is seen as a low priority as they would have to be created by programmers who themselves would see little benefit from them. As a result, software immigrants have to fend for themselves, and manage and create their own training, all while being asked to fulfil maintenance requests.

3.7 Conclusion

The issue of the maintainers fresh to the group, the software immigrants, is one that seems neglected by all but one of the companies in which I conducted interviews. Given that software immigrants would certainly need to gain a level-of-understanding about a sub-system before being able to perform useful work upon it, it seems strange that more consideration has not been provided to this end. Whilst the nature of the companies and teams surveyed seems very similar in nature to the companies of the Singer study, the Singer study does not specifically address the issues of software immigrants. This suggests that further reading is required to identify the type and volume of research that is focused on software immigrants and the problems, and solutions to those problems,

that they face. If this further reading demonstrates that the lack of training identified is as common as it would appear to be, then this suggests that any further work should be focused on examining approaches to try and address the lack of action by companies. Principally, I would be looking at approaches to help develop software immigrants' level-of-understanding that rely only on having access to the source code with no assumptions about the existence of external documentation or mentors. This challenging situation is the worst case scenario for the software immigrants and by providing an approach that works in such an environment then this can be augmented with other information sources if they are available in practice.

Chapter 4

Software Immigrants

4.1 Introduction

This chapter defines and examines the class of software maintainers known as software immigrants. What literature that can be found about them is examined and analysed, with particular attention paid to issues of staff turnover as well as formal and informal methods of training.

4.2 Motivation

Chapter 3 raised several interesting issues. Of particular interest were the actions (and inaction) that teams take to teach new hires (software immigrants) about the system they are going to maintain. Only one company had anything approaching a formalised teaching methodology, with the majority of companies using an informal mentoring system. Although software immigrants form only a small part of the overall picture of Software Maintenance, the interviews suggest that due to the specialising nature of maintainers, software immigrants are brought in to replace specialists who are the only source of information and useful work on particular parts of the system. As a result, it becomes important that the software immigrants gain knowledge of their area of responsibility as swiftly as possible.

As a result a further literature review was considered appropriate to try to identify what research had be done on the issues surrounding software immigrants, and to compare it with the results of my own interviews.

4.3 Identification of Literature

Trying to examine issues pertaining to software immigrants is a difficult task due to the small amount of material available. By reusing the literature base developed for the systematic literature review in section 2.5 a small body of literature was identified. For a paper to be considered to be about software immigrants it needed to have only a fleeting mention of them. This is in contrast to the Maintainer Focused category from

the structured literature review where a paper need to have significant discussion about the People aspect of Software Maintenance to be considered. The identified papers are presented in chronological order in table 4.1. The papers titled in bold are papers that are primarily about issues pertaining to software immigrants. The other papers are studies that are about other aspects of maintenance that happen to touch, sometimes only very briefly on issues pertaining to software immigrants.

Title	Author	Year	Reference
Help, I have to Maintain an Undocumented Program	Sandra Fay, Denise Holmes	1985	[23]
Applying Instructional Systems Development To Software Maintenance Education	Ronald Backus	1988	[4]
An Investigation into Software Maintenance – Perception and Practices	Paul Layzell, Linda Macaulay	1990	[41]
Delphi Study of software Maintenance Problems	Sasa Dekleva	1992	[18]
Software Maintenance Training: Transition Experiences	Thomas Pigoski, Steve Looney	1993	[54]
Beyond Program Understanding: A Look At Programming Expertise in Industry	Lucy Berlin	1993	[6]
A Change Analysis Process to Characterize Software Maintenance Projects	Lionel Briand, Victor Basili, Yong-Mi Kim, Donald Squier	1994	[9]
An Examination of Software Maintenance Practices in a US Government Organization	Alan Brown, Alan Christie, Susan Dart	1995	[10]
A Documentation Suite for Maintenance Programmers	Frank Cioch, Michael Palazzolo	1996	[15]
The Ramp-Up Problem in Software Projects: A Case Study of How Software Immigrants Naturalize	Susan Sim, Richard Holt	1998	[63]
Training for Software Maintenance	Mark Taylor, Eddie Moynihan, Andy Laws	1998	[74]
Evaluating the Predelivery Phase of ISO/IEC FDIS 14764 in the Swedish Context	Mira Kajko-Mattsson, Anna Glassbrook, Maria Nordin	2001	[35]
Extreme Maintenance	Charles Poole, Tim Murphy, Jan Huisman, Allen Higins	2001	[55]

Table 4.1: Software Immigrant Papers

4.4 Defining the Software Immigrant

Sim and Holt [63] are the ones that coin the term software immigrant. Their rationale is that:

> Joining a software development team is like moving to a new country to start

employment: the immigrant has a lot to learn about the job, the local customs, and sometimes a new language

They performed an exploratory case study that examined four software immigrants as they started work on a new system. The phrase software immigrant is used not just because of the connections between starting a new job and emigrating but also to avoid the use of the word 'novice'. Novice is a very loaded word, and in software engineering literature is most commonly used to refer to people who are learning to program. Berlin [6] uses the term 'apprentices' to label those who Sim and Holt termed software immigrants. Berlin, however, was specifically looking at programmers who were also learning a new language as well as the other identified aspects of being a software immigrant. The software immigrants studied by Sim and Holt had up to three full years of full-time programming experience. The apprentices studied by Berlin each had four years experience of application development in other languages. As a result the traditional use of 'novice' is ill-suited to describing the types of problems that software immigrants encounter. I have settled on using the term 'software immigrant' as I feel it clearly distinguishes itself from 'novice', whereas 'apprentice' still has some of the connotations that I am trying to avoid.

4.5 Training

An unwritten assumption in software engineering is that mentoring, where a senior programmer acts as a software immigrant's first port of call for help, advice and possibly even task assignment is, if not the best way, then at least the most accepted way of teaching software immigrants about a system. A key feature identified by both the Berlin and the Sim and Holt studies is that mentoring is an effective way of passing on information for the software immigrant, but is also time inefficient for the mentor due to the amount of work time the mentor is giving up while talking to the software immigrant. Mentors are providers of design rationale that is otherwise missing due to the lack of accurate documentation, but often the mentor works from the bottom up, reconstructing the rationale from the available code rather than simply remembering what it was. In the Sim and Holt study, software immigrants were given low priority maintenance tasks to help them develop a level-of-understanding about the system. Dekleva [18] performed a Delphi study with people involved in software maintenance, primarily maintenance managers. A Delphi study consists of first getting the group to identify issues of concern and then over a series of iterations rate those areas on a scale (in this case, 1 to 10). An iteration consists of mailing out a questionnaire asking respondents to rate each problem area as well as identifying further problem area. They also get to see the groups' mean for each problem area and are also asked to provide justification for deviating from the mean by three or more points. The purpose of the study was to try and reach a consensus between the participants as to ranking the lists of problems. The participants in this study specifically identified lack of programmer training as a problem, although it was ranked only 13th out of a list of 19 problem areas. This was the only specifically software immigrant related problem although documentation quality also featured as the 4th biggest problem.

The literature suggests that that there is a lack of implemented formal training for software immigrants. Taylor et al. [74] surveyed 31 British companies with the intention of examining and documenting their training methodologies. However, they found an almost total absence of formal training approaches: one company sent programmers on a training course that involved regression testing which was the only mention of maintenance activity; three companies (all financial service companies) encouraged the gaining of business qualifications, presumably to help them gain domain knowledge; while two companies seconded maintenance programmers to user departments to give the programmers a more well rounded view of how the software Products were being used. Only 12 of the 31 had specific maintenance-related technical standards in place, and even these standards were fairly minimal. An example given was that the maintenance standards:

> "consisted purely of a maintenance request specification, a test plan and test case procedure, and an authorization to go live procedure."

Layzell and Macaulay [41] performed a general maintenance survey of five major U.K. based companies. Their section on training highlights the lack of formal approaches to training and they discuss the theoretical, that there *should* be a systems encyclopaedia, maintenance personnel *should* have marketing skills, without identifying any of their companies that are *actually* doing that. There's no comparison or evaluation of these suggested techniques being implemented in practice. Kajko-Mattsson et al. [35] examined pre-delivery actions that impact on maintenance at eight different Scandinavian companies. Two of the specific issues they examined were the existence of maintenance plans and formal maintainer training. The maintenance plans that existed were at varying levels of formality and only maintenance plans for corrective maintenance were guaranteed to exist. Less than half of the organisations provided formal training plans for the maintainers, whilst the other companies have informal succession management style practices.

Briand et al. [9] present a paper about characterising the software maintenance process, in which they provide a case study focused on a particular software system. In this study they note that software immigrants are given basic maintenance tasks as a way of learning about the system, as no system documentation exists. There is a single head of maintenance who has been working on the system for a long time and who embodies all the system knowledge. With the loss of the head of maintenance all knowledge of the overall structure and design rationale of the system would be lost. He acts as a group mentor figure, to whom system questions are addressed. Poole et al. [55] presents a lessons learnt paper on the successful introduction of Extreme Programming principles to the maintenance of a company's core Product. In the main, the paper discusses the benefits of introducing a defined process as opposed the the entirely ad-hoc approach they were formerly using. However, of interest to the analysis of software immigrants is that the concept of pair programming is intrinsic to XP practices. They mention that one of the benefits of pair programming is that it makes more explicit the bond between mentor and software immigrant, although that is not the primary reason for pair programming. Beyond this semi-formalisation of the mentor–immigrant relationship they mention no other formal or informal learning or teaching practices.

There are only two common features identified from the literature. Firstly, mentoring is an unofficial industry standard for teaching software immigrants. Knowledge is passed on by face-to-face communication, with a collaborative question-and-answer style approach. The second is that software immigrants are given low-priority maintenance tasks to perform as a form of learning. On the basis of this evidence, it seems that there is little in the way of formal training for software immigrants. The following section discusses those papers that describe detailed, formal, teaching methodologies

4.6 Formal Approaches to Training

Backus [4] presents a very formal method of training constructed using the Instructional Systems Development (ISD) approach. This model involves the programmers sitting exams about the system they have to maintain. It is a holistic style of teaching and covers not just the Product to be maintained, but Processes that should be followed to maintain the Product. There is a mix of classroom and hands-on experience with the Product code. Compared to basic mentoring, in which the mentor's knowledge is assumed to be current, this methodology requires maintenance in order to remain relevant to evolving system functionality. In a mentoring environment the mentor's knowledge would assumed to be current, given that they are working with the current system. This methodology is quite rigid, making it easy to teach, but reducing the scope for the typical mentor-immigrant exchanges and the customisation of information to the specific need that mentors can give. Given the lack of references to the paper, and the general lack of structured teaching of maintenance discovered by Taylor et al. or Layzell and Macauly, this suggests that this formal approach has not found much favour with companies in the subsequent years. In many regards it is similar to having up-to-date documentation: it is something that people want to have, but without strong management will it will quickly fall by the way side. Indeed, Singer et al. [64] observed that while management would like up-to-date documentation, they do not think it is a worthwhile investment of time for expert programmers. Another negative aspect is that ISD is geared, with its upfront construction costs, to giving a view of the whole system. Given maintenance programmers' tendency to specialise on sub-systems the ISD based approach may well spend time teaching subjects information they will not use in their day-to-day activities.

Cioch et al. [15] also present a formal method of helping software immigrants gain a sufficient level-of-understanding to successfully maintain Products. Its major focus is on the idea that that as immigrants gain a greater level-of-understanding about a Product, they will require different information to more effectively learn about the Product. Whilst the system expert can pull out the requirements or design specification to find out what they want, such documents will contain too much information, presented in an ineffective format for the system novice. Their solution was a documentation site formulated at four different levels: newcomers; students; interns; and experts. The first two levels, newcomers and students, are considered non-productive stages, in that the immigrants would not be producing actual changes to the system. Newcomers is an initial orientation phase lasting only a few days at most. It mainly consists of giving software immigrants the cor-

rect *context* so that they can correctly align and associate information about the system they are working on. The three issues that should be covered are: "How you fit into the organisation"; "What is the purpose of the sub-system you are working on"; "Why the sub-system was designed this way". This information is to be presented in short, marketing overview type presentations. The student level starts covering "How the sub-system operates". This is done using the story telling approach [12] where students are walked through what the sub-system does when a specific, but common, action is performed in the system. This documentation does not exist fully formed. Students are instructed to construct their own story walk-throughs, in a similar manner as to how the programmers from chapter 3 stated how they learnt in a mentorless environment. This is to get the students to learn about the system in an active, rather than passive way. Intern level immigrants are shown "Organising and running to-be-released code" and "Process details". Process details covers company wide policies and standards along with the details of how their specific groups operates. Organising and running code covers testing methodologies, compilation, build workflows: effectively anything that is involved in working with the code that is not the actual code itself.

There are some unanswered questions about this approach. The paper talks in terms of a documentation suite, but constantly refers to the software immigrants as the ones producing the documentation. It is unclear how much of this is to be prepared by system experts and how much is self generated. The diagrams produced for students, which were hand crafted by system experts, are comparable in information presented to various UML diagrams. Given the availability of UML generators, experts would no longer have to spend time producing the diagrams, as they simply have to select from a plethora of automatically generated diagrams. The paper is also unclear on how long a software immigrant will be a student before moving onto the intern level. No means of measuring the suitability of changing the immigrant's status is stated apart from determination by a supervisor. Finally, frequent mention is made of supervision without stating how close it should be or how much time it should take up. So, despite initially appearing to present a highly formal approach like the Backus paper, this method relies on many subjective judgement calls by mentor like figures. This is not a criticism of the approach but it shows how hard it is to provide a flexible yet formal method of teaching, and shows the high value of system experts, mentors, in the learning process. They point out that there is a high upfront cost with producing the materials but do not discuss the problems of maintaining the materials to keep them current.

Brown et al. [10] examined the operations of a large US government organisation that developed and maintained multiple software systems. Systems were produced in an environment where one team created the system, while another, separate team maintained the system. The standard dysfunctions were found (poor communication, schedule pressures, systems not designed for maintenance, etc.). Of note for the study of software immigrants was that the organisation made heavy use of short term contractors to do work who had to be taught about the system. There was no unified or structured thought given to how to teach the contractors, and even a line of thought which pushed against giving them a wider contextual view (taking the opposite approach to the work of Cioch et al. which

started with giving the wide system overview) that hoped to obtain higher productivity by focusing the contractors on their small portion of the system without them getting distracted by the big picture. Given the whole ethos of poor management that the organisation had, this does not seem to be a fruitful approach. The government organisation, with its policy of high turnover, is an example of a organisation that would benefit from more formal training methods, but the tight schedules that it runs under precludes the creation of the necessary materials without a strong managerial will to change.

Pigoski and Looney [54] present an account of setting up a maintenance group from scratch. This is a group consisting solely of programmers who had not worked on the given system before, and in most cases had not heard of it up until the point where they received the code. Their techniques are based on the general advice of Fay and Holmes and can be summarised as:

1. Understand the Domain
2. Learn how the system is organised
3. Determine what it does
4. Practice by fixing low priority maintenance requests

As can be seen, the use of low-priority maintenance tasks is once again a feature of the learning process. Of particular interest, that is, going against common belief, is that Pigoski and Looney state that reading out of date documentation is useful, as even though it does not describe what the system is currently doing it can provide valuable background information. One possible reason for this might well be that they were building a department which would be given the responsibility of maintaining the software, as a result the Product was 'young' and the drift from what the documentation said and what the code did could well be smaller than for a system that has been maintained for five years. Another possibility is that their maintainers could very well have found the more abstract high level documentation useful, information that programmers trust more in documentation [64] than documentation that provides detailed descriptions of the code. To perform step 3, programmers were asked to read the code and then give oral presentations to their sub-groups as to what the code did. This technique reportedly had its own learning curve: talking about code is tricky, but proved to be invaluable in the long run. Finally there was learning by performing fixes. These fixes were performed before the code was officially their responsibility, so they were getting a headstart on the work. The Pigoski and Looney situation is slightly abnormal in that they had assembled an entire department who had fundamentally no knowledge of what they were about to maintain. There were no mentors to consult. This could well be another reason why the documentation was found to be valuable, as apart from the code it was the only store of knowledge about the system that existed, no matter how inaccurate it was.

Pigoski and Looney references the Dekleva study but draws from it a skewed conclusion. They state that the Delphi study identified high turnover as one of the principle problems of maintenance. While high turnover *was* identified as a problem area it was ranked *last*,

with a mean score of 3.9 and a standard deviation of 2.6 on a 10 point scale. This suggests that turnover is *not* a significant problem for the majority of respondents, but the high standard deviation suggests that for those respondents who did find it a problem, it was one of their most significant problems. The low rating is backed up by the Singer et al. study that showed maintenance personnel in their studied companies had a low turnover rate with high average system experience. The bimodal nature of staff turnover is also highlighted by the Leintz, Swanson, Tompkins derived studies. In a similar manner to the Dekleva study, although Turnover is identified as one of the top 10 problems by the LST study (ranked 9th) it has the highest standard deviation of all the responses. In the breakdown of answers Turnover has the third largest number of "Not a problem" responses (the lowest possible categorisation) with only the problems ranked 23rd and 24th (out of 24 problem areas) having a larger response in that category. The reason Pigoski and Looney highlight the problems of high turnover is due to the identified abnormality of their situation: effectively 100% of the staff working on the system turned-over simultaneously as the system moved from the development to the maintenance group. The government organisation studied by Brown et al. is an example of a maintenance group that *is* badly affected by high turnover, and shows how debilitating the lack of appropriate teaching mechanisms can be.

There are two inter-related reasons for a lack of implementation of formal training methodologies. The first is the large up-front cost in producing the formal materials that will not be cost effective in a traditional low turnover environment. Simultaneously, materials produced for formal teaching methods will have to be kept up to date with the system, and, given the traditional poor quality of systems documentation, this represents a significant ongoing investment by the company. That said, formal teaching methods are not a dead end by any means. It seems that companies with a high turnover should benefit from adopting more formal methodologies. Out-sourced maintenance departments, who focus solely on maintaining other people's systems, would certainly benefit from formal methodologies, especially as the informal approach of day-to-day, face-to-face discussion between software immigrants and mentor is often not available at all. The Pigoski and Looney methodology sits somewhere between formal and informal. The actual activities it sets out are of a fairly informal nature, they mostly consist of working with the code which is an informal learning activity. However, they are organised in a formal process with progressive steps to be followed. Therefore it is a formal process where each step consists of an informal practice. Also of note is the length of time spanned by the literature. There is a gap of 10 years between the Backus paper, detailing a potential formal training methodology, and the Taylor et al. study which highlighted the lack of formal training methodologies in practice. While there is an identified lag in adoption of research in Software Engineering, after 20 years it would be expected that at least a measurable percentage of firms would be adopting formal training methodologies if they were considered useful.

4.7 Comparisons with Interviews

These findings about software immigrants mesh well with the results of the interviews of chapter 3. The software immigrants literature suggests that software maintainers specialise in particular sub-systems to the detriment of the knowledge of the rest of the system. This is consistent with the results the interviews found. The literature discovers minimal, if any, adoption of formal teaching methodologies; from the interviews only one company out of seven had a formal methodology. The literature, contrary to common knowledge, suggests that for the majority of companies high turnover of maintenance staff is not a problem; none of the interviews put forward the information that there was a problem with turnover of staff. However, the literature does promulgate the belief that a mentor will always be available, something which the interviews showed is not always the case.

4.8 Conclusion

The existing literature on software immigrants consists of two types: one examines or mentions informal training methodologies; the other provides descriptions of formal training methodologies. While informative, these papers do not present any *comparison* or *evaluation* of these methodologies. While mentoring is consistently identified, there is no comparison between it and other approaches. In short there is no examination of the empirical difference in using different methodologies to help software immigrants gain a level-of-understanding about a system. Furthermore, with the exception of Pigoski and Looney there is a universal assumption of the existence of a mentor for software immigrants to consult, but, as my interviews discovered, mentors do not always exist, either due to company culture or unexpected events.

Given the lack of comparisons and the incorrect assumption about the existence of mentors it seems worthwhile to preform some sort of comparison between non mentor based approaches to building a level-of-understanding about an unfamiliar system. Due to the difficulty of performing a longitudinal, interventionist study in industry it was thought most appropriate to perform a controlled laboratory experiment. Specifically looking at *work based* approaches to software immigrants developing a level-of-understanding about a system. The following two chapters describe the experimental methods necessary for performing such an experiment as well as the design, implementation and results of the experiment.

Chapter 5

Experimental Methods

5.1 Introduction

This chapter is a guide to the construction and running of controlled software engineering experiments. Much of the advice is generally applicable to software engineering experiments as a whole, based on my experience and studying relevant literature, but is specifically focused on experiments dealing with level-of-understanding experiments. The chapter starts by describing the general problems of typical software engineering experiments, showing why they are more difficult than standard experiments. It goes through the elements required to construct, run and correctly analyse the data gained from a controlled laboratory experiment.

5.2 Nature of Level-Of-Understanding Based Software Engineering Experiments

As Software Engineering is the study of People applying Processes to Products so too are experiments in Software Engineering. This adds extra problems over experimentation in other fields. As recounted by Carver et al. [14], general experiments in the social sciences look at People and Processes. In the field of material sciences researchers examine Processes applied to Products. In general, Software Engineering blends these three components together to form a more challenging experimentation environment.

There are three general, basic designs of Software Engineering experiments, summarised in figure 5.1, which are based around the variation of each of the three elements of software engineering: People, Processes and Products. Type One experiments are where subjects in different groups use a different Process to try and gain a level-of-understanding about the same Product, an example of which is my own experiment from chapter 6. In Type Two experiments, all subjects are trying to perform the same task on different Products, the Products being created to match the same specification but constructed in different ways due to the Processes applied to them. This applies to experiments like Oman and Cook [52] where the same code was formatted in two different ways. This a particularly complicated combination as the Subject's performance on the task is being measured, but

it is quality of the Process used to create the code that is being determined by the experiment. Type Three experiments are when the People change but the Process and Product remain the same. This is the experiment design behind all novice/expert comparisons.

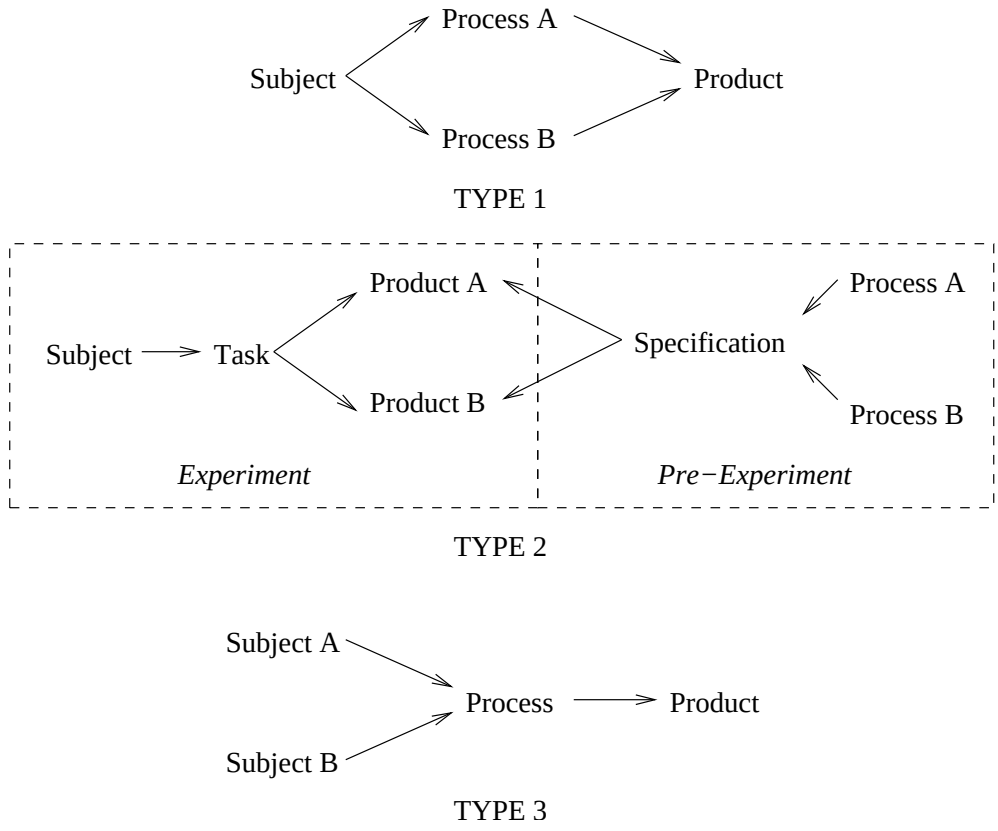

Figure 5.1: The Three Experiment Types

These designs are based around the idea of controlling the experimental environment. The People, Processes and Products are all termed as *independent variables*. That is, each of these things (variables) can be altered without effecting the values of the of the other variables. For example switching the People from novices to experts does not necessitate changing what Product they are working on. One aspect of experimental design that these diagrams do not cover is the measurement of the results of the experiment, i.e. what is known as the *dependent variables*. The appropriate selection and measurement of dependent variables is discussed in section 5.5. A given set of values for an experiment's independent variables is termed a treatment, factor or condition.

5.2.1 Hypothesis Construction

Empirical laboratory experiments are normally based around disproving what is known as the *null hypothesis*[85]. The null hypothesis is a statement of what is expected to happen when there is no measurable effect from altering the values of the independent variables. An alternate hypothesis should be given which describe the expected outcome of the experiment. Stating these hypotheses before the experiment is critical to the integrity of the experiment, as they affect how the statistical analysis of the experiment can be

performed. Producing hypotheses after the experiment is performed could easily allow fallacious conclusions to be drawn. Another important consideration when constructing hypotheses is the *direction* of the hypotheses. A hypothesis can be either uni-directional or bi-directional. That is, for an experiment featuring two treatments, the experimenter can test for only one treatment being superior to another (uni-directional) or for either treatment being superior to the other (bi-directional). For example if the null hypothesis for an experiment is that *subjects performing treatment A will have no difference in their level-of-understanding from subjects performing treatment B* and the alternate hypothesis is that *subjects performing treatment A will have a higher level-of-understanding than subjects performing treatment B* then this would be a uni-directional hypothesis and when performing statistical analysis the experimenter can only test to see if treatment A is better than treatment B. If the experiment shows that treatment B is better, then that is not a significant result. The reason an experimenter would choose to construct an uni-directional hypothesis is that it allows the measured difference between the two groups to be smaller when testing for significance as the experimenter is only looking for a difference in one direction.

5.3 People

This section examines the two most important issues in selecting subjects: subjects' ability levels and ethical considerations for selection.

5.3.1 Ability Levels and Professionalism

A frequent criticism, considered in depth by Curtis [16], of software engineering experiments is the constant use of undergraduate students rather than industry practitioners. The reasons for both the criticism and why student programmers are still used is fairly obvious. Critics do not feel that undergraduates accurately represent general programmers and thus affect the generalisability of the results of an experiment. However, researchers have far easier access to undergraduates than industry practitioners, which results in controlled laboratory experiments frequently using undergraduate subjects.

The principal advantage of using student programmers in experiments involving programming is that the students will have undergone long term assessment of their programming ability. As programming ability is highly variable amongst practitioners and variation in programming ability is the single largest confounding factor in any experiment involving programming, there needs to be some accurate way of stratifying or blocking the subjects based on programming ability. As stated, students will have had numerous tests of their programming ability over their time at university. Using the assumption that these assessments are reasonably accurate, this allows the experiment to confidently balance groups.

Even with the greater level of homogeneity that using continually assessed undergraduates brings, there are still problems. In a rather unscientific study, Spolsky [69] analysed the work of Yale students and found that the best students in the class were around five to ten

times better than their peers. In industry this variation becomes even more pronounced, with Boehm [7] recognising the best programmers as five times better while Sackman [59] records the best programmer being up to 28 times better. There is no reliable way of judging programming ability in a short amount of time, making it very hard to guarantee the homogeneity of the subjects when using industry practitioners.

For industry practitioners, experience alone is not a good judge of ability, as recognised by Vokac et al. [79] and Jørgensen and Sjøberg [34]. Pre-testing is also not a completely adequate approach. Vokac et al. found no correlation between performance on the experiment pre-test and performance in the experiment itself. This demonstrates that there is no good, quick way to test for programming ability, even when the facet of ability is quite specific (as in the Vokac et al. case). Vokac et al. suggest using longer calibration tests but do not offer an opinion on how long they should be. A year long course (in the case of students) gives an accurate indication of ability, but how long professionals should be given is an open matter. Furthermore, the longer the calibration test the longer the experiment takes to perform and thus the harder it is to run the experiment at all. One approach that does seem to effectively balance groups is manager review [3]. By asking managers to rate their programmers on a three interval scale (Good, Average and Poor), the groups were sufficiently ability balanced. Further studies would have to be performed to make sure that this method of categorisation is sound and not just down to particularly good managers. Perhaps combining manger review with peer review, where subjects rate themselves and other subjects, would help in producing an overall ranking. Thus, by using multiple imprecise sources of information, the experimenter can try to find the Venn-like overlap in the information, or produce an average score that best categorises each subject's ability.

5.3.2 Ethics of Inducement

There are a number of methods of obtaining subjects for experiments. The most common for student subjects are: asking for unpaid volunteers; paying or remunerating volunteers; awarding course credit for experiment participation; or making experiment participation a compulsory part of the course. Each has its own set of issues, but the one that I have most concerns about is making experiment participation a compulsory part of the course. There are reasons for adopting this approach, the primary one being that the experimenter can guarantee themselves a large pool of subjects – often a great impediment to running experiments – and there are also benefits when it comes to training the subjects which I will discuss in section 5.4.3. However, I have serious ethical, and methodological, concerns about compulsory participation. I do not think it is morally justifiable to force students to take part in an experiment, especially if their performance in the experiment translates directly into coursework marks. This is particularly true of between-groups experiments – which, as I will explain in section 5.4.1, are essential for many software engineering experiments – as some students would have to do different work to others, and as such may be achieving lower marks. Trying to solve this problem by making the experiment worth nothing to the overall course grade, such as in Thelin [75], introduces a large motivational issue. Students are now told that they must produce a piece of work

but that the quality of work is completely unimportant. Although I have no experimental data to back this up, I would imagine that the work produced by any such subjects would be of a lower quality than that of either subjects for whom the quality did relate to their course grade or volunteer subjects.

In the case of professional programmers, there is much less literature on how to induce them to perform laboratory experiments. The Simula lab [67] pay companies to supply developers for some of their experiments, often they hire consultants, so that for the subjects it is like any other paid day's work. As they are not doing their day-to-day work, it might be seen as similar to a paid holiday, and so a question remains as to whether the subjects will treat the experimental work as seriously as, or more seriously than, their normal work. However, there is no evidence to suggest that either would be the case.

5.4 Mechanical Construction

5.4.1 Between and Within-Group Experiments

In standard experimental design, within-groups experiments are superior to between-groups experiments [58]. In an ideal within-groups experiment, the subjects perform all the conditions of the experiment, for example they perform Condition A, which is to debug a program with a tool, and then Condition B, which is to debug the program without a tool. Half the subjects would do Condition A then Condition B while the other half will do B then A, in order to reduce the effect that the ordering of the tasks has on the results. In a between-groups experiment, subjects do only one of the conditions and the relative performance of the groups is analysed. Within-groups experiments are generally considered superior because it eliminates subject ability variability from the experiment results, as it guarantees both conditions have equally able participants since they contain all participants.

However, with many types of software engineering experiments, and with all experiments that involve understanding code, the ideal within-groups design described above is impossible to perform. You cannot get a subject to learn about a system one way then learn about the same system in another way whilst not benefiting from performing the first condition. They have learnt about the system the first time, and they cannot "unlearn" that knowledge. As a result, in level-of-understanding experiments, only between-groups experiments can be performed and the treatments groups must be carefully balanced for ability.

There are a number of experiments (for example the repeated software inheritance experiments [17, 13]) which follow a pseudo-within-groups design for types 1 & 2 experiments (see figure 5.1). There are two subject groups that perform two conditions each in a standard fashion. However, due to the problem of the learning effect identified above, rather than simply changing the Process the subjects perform, the experiment also changes the Product the subjects use as well (see table 5.2). This eliminates the learning effect issue, as two different Products are being used, but immediately renders it very difficult to draw

conclusions as if the experiment was a true within-groups experiment. When two factors have been changed simultaneously (both the Process and the Product), you cannot compare a subject's time on the first treatment with their time on the second treatment, as the Product (which is a critical part of Software Engineering) has changed, unless you can show that the two Products are identical. However, if the Products are identical then there would be some form of learning effect in operation. So you can compare results horizontally across the table (comparing Process A to Process B when applied to Product X) but you cannot compare vertically (comparing Process A applied to Product X to Process B applied to Product Y) and ascribe all of the reason for any difference to the change in Process.

Process A & B −− Product X & Y

Group 1	Group 2
Product X Process A	Product X Process B
Product Y Process B	Product Y Process A

Figure 5.2: Pseudo Within-Group Experiment

5.4.2 Comparing Like with Like

In a related topic, it is important for experimenters to carefully consider what is being changed in different treatments. By this, I mean that the experiments must conform to the patterns in figure 5.1 without adding any additional, unaccounted for, factors. For example, in the Rigi/Shrimp experiment [71], the authors claim that they are comparing the use of Rigi/Shrimp versus using **Sniff++** to understand code. That is, they are conforming to experiment Type 1, with specific structure as shown in figure 5.3. In actual fact they had made a slight change as summarised in figure 5.4. The experimenters had modularised the code into logically connected chunks and given them useful names: a form of packaging. This was not done automatically by the tool, but by someone who was familiar with Rigi, Shrimp and the program used in the experiment. As a result, the experiment was now a comparison between using **Sniff++** to examine a program versus using a tool to examine a program that had been modularised and restructured. This is not comparing like with like. This does not invalidate the experiment, but the hypothesis must be restated to include the modularisation of the code, as at the moment beneficial effects of the modularisation are being ascribed entirely to the tool. If they did not wish to change their hypothesis then additional information could be provided to the **Sniff++** group, for instance by providing a textual representation of the modularisation made for Rigi and Shrimp. They make the same type of modification in another similar experiment [70], attributing external factors to the tool's benefit.

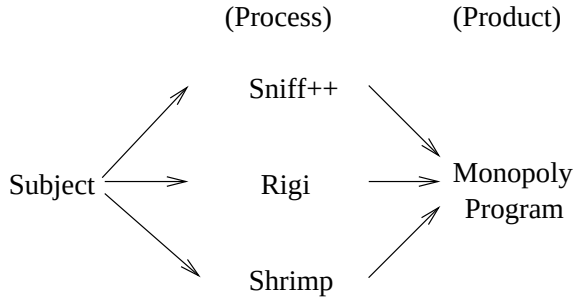

Figure 5.3: Stated Experiment Design

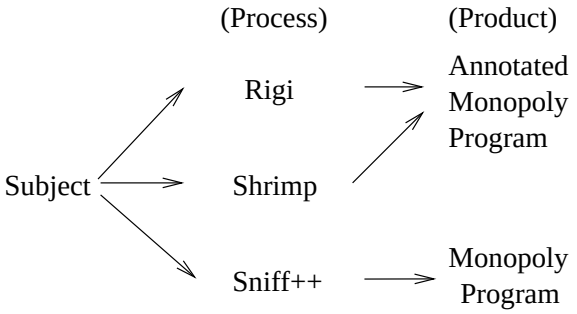

Figure 5.4: Actual Experiment Design

5.4.3 Training the Subjects

Possibly the most vexing question facing experimenters is how to give appropriate levels of training to subjects. Even when using professionals there is no guarantee that they will be experienced with the specific area of Software Engineering that the experiment is covering [67]. Often subjects are often being asked to perform a technique they are not familiar with, or use a tool that has been recently developed. This is a vitally important issue and it is one where student subjects have a significant advantage. By using course time to teach subjects how to perform a particular technique (say a code inspection technique), the experimenters can make a better attempt both at guaranteeing a consistent level of teaching and balancing the groups for different ability levels. It is well known for tool use that in the initial usage period, programmers adopting a tool become less productive as they learn the intricacies of the tool [27]. In a typical controlled experiment the experimenter has exactly this problem, as they may have only an hour at most to train the subjects on the use of the tool before performing the experiment.

Training time is not the only teaching related problem: another is accidentally training the subjects to use implicit problem solving techniques. If the experimenter wishes to ascertain how useful Tool X is at diagnosing a type of bug (say a memory leak), the simple experiment would have half of the subjects trained up to use the tool, and the other half would use no additional tools as a control group. However, in training to use the tool there is also implicit training to solve the problem the tool is addressing. Because a within-groups approach is not possible in this situation (section 5.4.1), if both groups

were to be trained in the use of the tool, then one group necessarily will have been given information that goes unused and expectations that go unfulfilled. This problem can be solved by organising an experiment as shown in figure 5.2 but it must be accepted that this is effectively two between-groups experiments and not a within-groups experiment, as subjects' performance on one program cannot be compared to their performance on another program. Furthermore, this increases the time for the experiment.

5.4.4 Fatigue Effects

Long experiments risk issues of subject fatigue, especially if the tasks required are mentally or physically strenuous. Fatigue refers to the deterioration of a subject's performance at stages in the experiment due to increased tiredness or boredom. This is of critical importance in within-group experimental designs as a subject's performance in later tasks is directly compared to their performance in earlier tasks. With between-group experimental design, fatigue effects are much less important to consider, as these subjects are not compared against themselves but against other subjects who would be suffering the same risk of fatigue. In within-group experiments some effort has to be taken to try and identify if subjects are suffering from fatigue effects. For between-groups experiments, unless there is special reason to believe that subjects will have endurance levels of such differing degrees that it will affect their experimental performance, in which case it would be an independent variable that must be controlled for, it can be assumed that any fatigue effects will be balanced across the groups by random allocation.

5.5 Measurement

There are several ways of measuring a programmer's level-of-understanding of a piece of code. Dunsmore et al. [21] examine many approaches which they then classify into four groups: maintenance, dynamic, static and subjective. To help compare the methods, it can be useful to think of them on an axis classifying the measurement from being direct to indirect as seen in figure 5.5. The subjective method, which consists on getting the subjects to rate their own level-of-understanding, is a purely direct approach to measuring a subject's level-of-understanding of a program: there is no process to interrupt or manipulate a subject's self-rating of their level-of-understanding. Another highly direct method, that Dunsmore et al. did not consider, would be for the experimenter to ask the subject to explain to them what specific parts of the code do. The experimenter, who would have a complete understanding of the code, can then rate their knowledge to produce an overall level-of-understanding score. Indirect methods cover getting the subjects to perform tasks that would be helped by a high level-of-understanding of the code. Swift completion of the task reflects a high level-of-understanding of the code, assuming you can balance for the ability to perform the task. Performing a maintenance task is an example of a purely indirect method, nothing about it relates directly to levels-of-understanding but given that to perform maintenance a certain knowledge of the code is required it follows that a high level-of-understanding of the code would allow a subject to complete the maintenance task 'better' than a subject with a low level-of-understanding. Static and dynamic represent answering written questions about the code. Static questions deal with program

structure, for example, what module a method is located in, or what functions a method calls, while dynamic covers questions about data flow and run-time characteristics of the code. On the sliding scale I classify these techniques as sitting at about the halfway point between purely direct and indirect techniques.

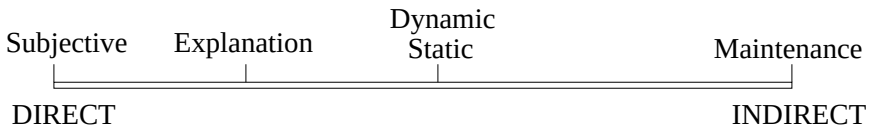

Figure 5.5: Axis of Methodology for Measuring Level-Of-Understanding

Using indirect measurements seems at first like an unnecessary confounding factor, but the more direct measures have problems as well. The ability to explain code to another person, while potentially highly desirable, is not a "required" skill of programmers, and is one that is difficult to quantify and thus control for. As Pigoski and Looney [54] reported, they had problems in getting experienced programmers to explain code to other people, many of them finding it unnatural and difficult. The manner in which information is solicited from the subject is also problematic, as poorly worded questions could well affect the types of answers given. Purely subjective ratings from the subjects are clearly problematic. Despite the statement in Dunsmore et al. that there was a reasonable correspondence between subjects' self-rating and their test scores, there is still considerable overlap between various self-rating groups. Furthermore, subjects were being asked to rate their level-of-understanding on a small program, but it may well be that subjects are less competent at rating themselves when faced with comprehending a large program.

Test questions are difficult to create and must be tightly coupled to the subjects' ability. Dunsmore et al. noted that when they used the same set of questions with more experienced programmers they encountered a ceiling effect[1], as these subjects were not troubled by issues that tripped up the less experienced programmers. Problems of repeatability are raised due to the need to tightly couple questions to ability level, as the same questions cannot be used between subjects of different ability levels.

The format of the test is also important. Multiple choice tests have numerous issues relating to the composition and ordering of incorrect answers [5] as well as the appropriateness of marking schemes that do not use negative marking. If free response questions are used then the type of expected answer must be defined, to avoid the possibility of subjects wasting all their time providing detailed answers to simple, low scoring questions. Obviously, if the tests have to be taken from memory then they become more a test of a subject's short and medium term memory rather than their level-of-understanding of the code. If tests are used then they should be performed open book, that is, with access to the code that the subjects are being quizzed about.

[1] That the more experienced programmers were giving almost all correct answers, thus making it more difficult to distinguish between them. There was a not a spread of results, instead a cluster of high values.

Finally, if both quality of answer and time to answer are used to measure the subjects' level-of-understanding then the results then become ratios which are particularly hard to statistically examine as they do not have a well defined mean, variance or standard deviation.

The ability of a subject to perform the task being measured is a confounding factor in experimental design. Ideally experimenters should be able to select a task does not rely on any great skill or relies on a skill that the experimenter can easily control. In this regard, maintenance tasks have a clear advantage, as the ability to develop, debug and enhance programs are the core skills of a programmer. As seen in section 5.3.1 it is possible to categorise subjects by programming ability and thus reduce the chance that ability levels affect the results of the experiment. Although Dunsmore et al. suggest that maintenance tasks are no better at measuring level-of-understanding than the other methods, their analysis of the experiment has two flaws which affect its generalisability. Firstly it was performed off-line, that is, subjects made changes to the code by making annotations on a print out rather than interactively at the computer. This could have adversely affected their evaluation of maintenance tasks as a measure of level-of-understanding as it is not reflective of real-world practice. Secondly, they used an incorrect statistical test for assessing the subject's performance (see section 5.8.3). As a result of the ability to closely and accurately control for subject ability, I believe that the most effective way of measuring a subject's level-of-understanding of a program is through making the subject perform a single maintenance task on the code.

There are multiple methods of determining the *quality* of the completed maintenance task, for instance neatness of code, correctness of code or efficiency of code. Neatness and efficiency measures both introduce subjectiveness to the results. Given that coding is an area in which the placement of the curly brace can inflame passions amongst otherwise rational people, it seems that introducing these subjective measures is a potential powder keg of uncertainty when it comes to attempting to externally *replicate* the experiments, as one person's neat code is a disgusting kludge to another. Correctness of code provides a clear unambiguous result: either the change works as specified or it does not. This measure also comes with an easy to analyse value: the time it takes to complete the task. It does not produce any difficult to analyse ratios nor does it rely on a subjective interpretation of results: there is no half-working for quantitative evaluation.

5.6 Materials

5.6.1 Code

In this section I discuss the issues surrounding the construction and use of computer programs and code fragments in experiment design. Much of what I write is applicable to all Products used in level-of-understanding experiments and so could equally well be applied to UML diagrams or code specifications, for instance.

Bespoke or Pre-Existing

There is an unwritten assumption that using existing, real world, code is better than constructing code from scratch for the experiment. There are numerous good reasons for using pre-existing code that can be resolved back into the issue of generalisability. Performing an experiment on pre-existing code makes the results of the experiment more generalisable and thus the results more relevant to researchers and practitioners. However, real world code is designed and written to solve a specific domain problem. As a result, issues of size, domain and complexity of the code necessary to perform the experiment can severely limit the choice of pre-existing code available. If a pre-existing piece of code is selected that has problems in domain, size or complexity then that risks jeopardising the results of the experiment by allowing these confounding factors to bias the results of the experiment. As a result, it may well be necessary for the experimenter to produce the code themselves, adhering to guidelines presented below.

Product Size

To allow the results of an experiment to be generalisable, experiments have to be performed on reasonable sized Products, be they computer programs, formal specifications or documentation. An experiment measuring some general quality is far more valid when performed on a program of 150,000 line rather than 15 line program. As von Mayrhauser and Vans summarise [80], many experiments work with programs of less than fifty lines of code. This is not to say that experimental results from experiment on "small" pieces of code are not valuable, but that they are less generalisable than experimental results produced from work on larger pieces of code. This is particularly the case in maintenance where large pieces of code are the norm where a conservative estimate would suggest a software maintainer is responsible for round 40,000 lines of code per system [44].

Despite the desire to perform experiments on large, and thus more realistic, pieces of code there is a competing demand of time. Experiments in software engineering will often have a time limit measured in a number of hours or even minutes rather than the weeks, months or years of other disciplines. The shorter the length of the experiment, the smaller the given Product has to be. There are no guidelines on how large *any* given Product should be for the length of time of an experiment. Intuitively, it will vary from experiment to experiment depending on what is being measured. Robinson [58] sees this as the type of issue that can *only* be resolved by piloting the experiment design with sample materials, measuring the result and refining the materials. From my own practical experience, final year university subjects can gain a level-of-understanding of a piece of code of approximately 1500 lines in length after performing general programming tasks on it for one hour to accurately answer questions upon it and/or successfully perform further task(s) upon it. The issue of code size should not be considered in isolation, as it is inextricably linked to that of code complexity.

Product Complexity

Programs acted upon by the subjects in an experiment should neither be unnecessarily complex, nor overly simple: they must sit at the awkward saddle point of being "rea-

sonably" realistic. Unless the experiment is measuring, for instance, how formatting can hide bugs or confuse programmers, then anything particularly clever or devious that goes into the construction of the code acts as a confounding factor. This, once more, is a common-sense part of Product construction: unless you are measuring a Product constructed due to an unusual Process the Product should not be unusual. Preferably there should be some way of measuring program complexity against an average subject's ability level. However, there is controversy over the general utility of program complexity metrics [38, 84]. Furthermore, there are no metrics that measure the effects of program formatting or identifier naming. As a result there are no metrics that can accurately measure the *overall* complexity of a given piece of code in a way that would quantitatively aid the construction of a piece of code for an experiment. As with program size this is once again an issue that can only be resolved by subjective judgement and careful piloting of the materials.

Domain

The domain in which the experiment is performed is a confounding factor. Referring to figure1.1, it can be seen that the basic knowledge layers are Technical, Domain, and System. A typical level-of-understanding experiment assumes that participants have a certain basic level of Technical knowledge and, hopefully, no knowledge of the System. Often the experimenter will know very little about any subject's knowledge of the domain used in the experiment. The selection of a domain where subjects could have wildly different levels of knowledge without then trying to level the knowledge can have a marked effect on ability of the experimenter to analyse the results of the experiment. Even worse than the split between no domain knowledge and full domain knowledge is misunderstood domain knowledge. Storey et al.'s [71] experiment used a program that aided play of the game Monopoly, but as there are many different commonly played house rules to Monopoly, two people who claim to understand the rules of the game might well be playing two different versions of it. Although the subjects were given some time to look over the rules, they did not have to prove any level of domain knowledge before beginning the experiment, so misconceptions could be carried into the main part of the experiment.

The ideal program domain has two qualities. The first is that it is a simple domain, which can be easily taught and understood in a short amount of time. Francel and Rugaber [25] got the domain selection exactly right when it came to simplicity. The concept of counting up the number of words on a line is an atomic concept and as such it is an easily communicable idea. There are some fine details to work out, such as if words are split between lines, but the entire domain can be taught in minutes if necessary. The second desirable quality is that of real-world relevance. The domain should preferably relate to a real-world concept that you would expect to have a computer based solution for. Once again, a word count program is entirely plausible as a real-world application, in that such functionality appears in almost every text editor ever written. Furthermore, a subject's knowledge of the domain should be tested to make sure they do not have a significant gap in their knowledge that would impair their ability to perform the experiment. In this regard, a possible improvement to my experiment detailed in chapter 6 would be

to give the subjects a test on the domain to make sure they all had a similar level of understanding of the domain, although like many improvements this does increase the amount of time needed for the experiment.

5.6.2 Comparability of Materials

Many software engineering experiments examine the differences between pieces of code that have been constructed in different manners: that is, they conform to the Type 2 experiment design. For example, one program might be constructed with multiple inheritance, while another program is constructed with no inheritance, and the experiment is designed to measure which program is easier to understand. It is important to determine if this is a legitimate comparison. It is my contention that the comprehensibility of a given program may well be related to a combination of the domain the program is modelling and the techniques used to construct it, with different problem domains more optimally fitting different program construction techniques. Furthermore, if the multiple different programs do not match the same specification then there is no evidence that they can be considered comparable at all. This is precisely the reason I decided not to use a corrective maintenance task for my experiment in chapter 6. Indeed, the repeated failed attempts at replicating program inheritance experiments [17, 13, 30] seems to suggest that different, but similar, programs are not strictly comparable. In the inheritance experiments, the basic format would see subjects try to understand one of three different programs. Program A would be flat with no inheritance, program B would have a "medium" level of inheritance, usually three levels, and program C would have "deep" inheritance, usually five levels. These three programs are created by writing one of the three programs first, say program B, then programs A and C are created by modifying B to get the desired levels of inheritance. I share the opinion of Deligiannis et al. [20] that this is not a good way of producing the programs. They may match the specification, but they have not been created in the best way for the inheritance levels used. By not being produced in the best way, programs A and C are going to be intrinsically harder to understand, as they are underpinned by bad design decisions. Despite the failures in replicating the results of previous experiments, none of the experimenters have indicated that the way in which the programs were constructed could be an issue.

However, experimenters still wish to perform controlled experiments to determine if program construction techniques have an effect on programmers' level-of-understanding of the code. I feel that if different construction techniques are used they should be embraced to their fullest rather than trying to force program to be "the same but different". By trying to force what should be widely different programs to be as similar as possible, experimenters are undermining their attempts to determine if different program construction techniques have an effect, as they are sabotaging the experiment code. Although the alternative is to have programs that are significantly different in nature, it has to be accepted that programs built using different methodologies *should* be significantly different. This is an approach used by Razali et al. [57], whose experiment to determine the level-of-understanding differences between B and UML-B specifications was founded on the creation of the materials from scratch rather than generating one set of materials from

the other.

5.6.3 Materials Conclusion

The materials used in an experiment have a number of competing demands that govern their construction. The first is that the Product should be appropriately sized for the length of the experiment, and only piloting of the experiment design can truly reveal if it is of an appropriate size. If a Product is too small, the results will lose generalisability, yet if it is too large for the time, the size becomes a confounding factor. Unless specifically examining issues of Product complexity, the materials should not be intricately constructed; they should be straight forward, otherwise they introduce a confounding variable. Likewise the domain the materials are in should either be very well known to the the subjects or be trivially easy to teach, otherwise varying levels of domain knowledge become a confounding factor. Finally, if separate treatments of the experiment require different or altered materials then this must be identified in the experimental design lest the effect that these different materials have be overlooked in the analysis of the results.

5.7 Subjects' Actions

5.7.1 Editing Code

If subjects have to edit the code as part of the experiment, for instance if the experimenter has decided that a maintenance task is the best way of measuring understanding, then the way code is edited becomes a confounding factor. A common approach is to have subjects annotate a paper listing of the code, which is unrealistic, to say the least. Programmers have multiple sources of feedback when editing code using a computer, all of which are removed when they have to unnaturally edit code on paper.

5.7.2 Lab Environment

The laboratory environment is a confounding factor. In general, programmers work in teams in communal areas. They are interrupted by a large array of external events, can take a coffee break if they choose, or consult other programmers if they are having a particular problem. A lab environment removes all of these factors. While this reduces the "real worldness" of the environment it also allows the experimenter to be more certain that any effects seen are due to changes in the independent variables.

5.7.3 Think-Alouds

A common approach in program understanding experiments is that of the think-aloud. Getting subjects to say what they are doing as they are doing it. Unfortunately, think-alouds also have an affect on how subjects perform their work as demonstrated by Hughes and Parkes[32]. Their experiment showed there was a significant difference in the work produced by subjects who worked normally and those who thought-aloud or who verbalised mental planning. Whilst think-alouds are a valuable way of gaining information for a certain aim (for example validating a mental model of program comprehension), it

would be an unnecessary confounding factor when applied to another experiment as a secondary goal.

5.8 Statistical Tests

There are a wide variety of statistical tests for measuring results from experimental design [85, 58]. While, in detail, they have a widely different approaches, in general they tackle the analysis of the data in broadly the same manner. The fundamental principle is that if there is no difference between the treatments in an experiment – that is, the null hypothesis is true – then there should be no difference in the results of the subjects. The wider the divergence of the results of two groups, the more chance there is that the treatments have caused this effect. The larger the volume of subjects in each treatment, the smaller the difference between two treatments has to be for there to be what is considered a significant difference. The test discussed below ultimately produce a *p-value*. To determine if a statistically significant difference has occurred, experimenters choose what is known as a significance level. If the obtained p-value is below the selected significance level the null hypothesis is rejected and the result is said to be significant at the selected level. The smaller the selected significance level the stronger the result.

The following sections discuss some standard statistical tests (t-test, ANOVA & Chi-Square) commonly used in software engineering experiments, as well as briefly discussing the two types of statistical errors that can be made. It also describes another common statistical method, Survival Analysis, that has seen practically no use in the software engineering literature but is of particular relevance when using only a single task as a measure – as I have done in my experiment detailed in chapter 6.

5.8.1 Statistical Errors

There are two types of errors that can occur when using statistical tests (apart from making mistakes during the calculation). They are called Type-I and Type-II errors [50]. A Type-I error is when the experimenter rejects the null hypothesis in favour of an alternative hypothesis despite the null-hypothesis being true. This occurs when the p-value from a statistical test is below a chosen significance level despite the different treatments having no effect – i.e. the result happened by chance. A Type-II error happens when the null hypothesis is not rejected despite it being false. This would happen when the results from a statistical test are above the chosen significance level despite there being an effect caused by the treatments, again happening by chance. It can be seen that raising the significance level reduces the chance of a Type-II error occurring but increases the chance of a Type-I error occurring.

5.8.2 Standard Tests, both Parametric and Non-Parametric

t-test

The t-test is an exceedingly common and fairly robust statistical test for measuring the statistically significant difference between two means. The standard parametric t-test

assumes that the two treatment groups are reasonably normally distributed. There are a variety of non-parametric variations on the t-test that allow it to be used with non-parametric data. In the standard parametric test used for a between-groups experiment there are three assumptions made.

1. The the samples are normally distributed
2. The samples are of equal variance
3. The measures are independent

The result from the t-test is the eponymous t value. The t value is the difference in the means of two groups divided by the estimated standard error of the difference between the two means. Potential t-values form a curve that is very similar in shape to the normal distribution. The greater the value of the t the greater the chance that the two sample groups are not drawn from the same population and thus the greater the possibility of being able to correctly reject the null hypothesis.

There are a number of limitations to the standard t-test. If there are more than two treatments then multiple t-tests would have to performed. Each t-test performed increases the chance of a Type-I error occurring. Bearing in mind that each additional treatment means an quadratic increase in the number of tests that would have to be performed, that would mean that with only five factors, with each test performed at the $p = 0.05$ level then there is in fact a 40% chance of finding at least one p-value of less than 0.05 by pure chance. The way round this is the use of the ANOVA test which combines all the treatments together in a single test. If the ANOVA test reveals a significant difference then judicious selection of treatments can be applied to find which precise treatments are causing the difference.

ANOVA

The ANOVA (ANalysis Of VAriance) test is used to discover what is known as the F ratio. The F ratio is the observed variation of the group means divided by the expected variation of the group means. If the null hypothesis is true then F would be 1, that is, the observed variation equals the expected variation. When this is not the case the larger the divergence between them, the greater the value of F. A large F value allows the experimenter to reject the null-hypothesis (of all means being the same). However, it does not allow the experimenter to say which mean or means are different, nor by how much. As stated above, the simplest way would be to select likely looking treatments and perform the t-test upon them, but one still cannot simply perform a t-test on all the pairs of treatment groups as that still risks making a Type-I error. Another approach is to calculate confidence intervals for the multiple treatments and try to find which means fall outside the interval groups. The best approach is to state the expected differences in the alternate hypotheses, which allows the experimenter to check for the expected difference. If an unexpected difference occurs then the null hypothesis cannot be rejected and new options must be considered.

Chi-square

The chi-square is a multiple purpose statistical test for examining categorical frequency data. It is mostly used for either comparing actual frequency distributions of a single treatment with theoretical distributions or comparing the proportional distributions of two or more treatments. In the first case, each measure has its difference to the theoretical value measured. When measuring the proportional distribution between two or more treatments, the observed values are first placed in what is known as a contingency table. A contingency table is a n-by-m grid where n is the number of treatments and m is the categorical results. Each cell of the table holds the frequency for one outcome of a single condition. A theoretical even distribution for each combination of variables is calculated, one that would be correct if the null hypothesis were true. The difference from this theoretical distribution is then calculated. The larger the difference the more likely the result is statistically significant.

A common use of the chi-square statistics is testing *goodness of fit* for numerical distributions. Goodness-of-fit is a term that covers all methods of checking if observed data matches a theoretical or expected distribution. This is commonly used for testing whether a distribution is normal enough to apply a t-test to. In this case, the data would be quantised into a number of bands (say 1-3, 4-6, 7-9, 10-12 and 13-15 for a 15 point scale) and compared against the theoretical frequency for a "perfect" normal distribution for the number of observations. The chi-square test will show if the observed data comes from a non-normal distribution. However, like all approaches that rely on quantising data, this approach is sensitive to how the data is grouped and is also subject to the general problems of the chi-square test detailed below.

The chi-Square test gives less reliable results as the expected values for each cell fall. It is generally agreed that if any expected value in a 2-by-2 contingency table is less than 5, the chi-square test can not be reliably used. In larger tables, it is considered that if more than 20% of the cells of the contingency table have an expected value of less than 5, the chi-square test should not be used.

5.8.3 Survival Analysis

Many experiments are performed under a time limit, that is, subjects are given a maximum amount of time to complete a given task or tasks and if they have not finished them in that time then they must stop. This introduces a cut-off point which affects the statistical tests that can be used to analyse the completion times. If subjects have been completing a scored test then they have a result (no matter how low) that can form part of a distribution (be it normal or non-normal). However, if they are performing a single task (for instance if the experiment design is using completion of a programming task to measure level-of-understanding as described in section 5.5) then there are subjects for which there is potentially no result as they do not finish within the given time. As they have no completion time they cannot be used in calculations of the groups' means or standard deviations, which is a critical step in the ANOVA and t-test, nor do they simply fit into a category for performing chi-square analysis. However, they cannot simply be

excluded as this introduces an obvious selectional bias to the experiment. Examples of this selectional bias can be seen in Dunsmore et al. [21], in which subjects are timed completing a maintenance task. Some subjects do not finish in time and they are discarded from the statistical analysis. In the paper, they comment on the fact that they have this right-censored data, although they do not use the term 'right-censored'. They use standard t-tests, but consider the possibility that their removal of the censored students will have affected the results, and they specifically state they do not know what the appropriate statistics to use in this case are. The correct branch of statistics to use would be survival analysis.

The general survival analysis technique I am examining is the use of Kaplan-Meier survival curves, using the log-rank test to examine the results [36]. The main fields of research in which survival analysis comes from are medicine and mechanical engineering, where they are used to measure *time-to-event* data, where the event might be, for example, patient death or component failure. The concept is generic and can be directly applied to any experiment that is measuring time-to-completion of a task. The key feature of survival analysis is that it allows computations on what is termed *right-censored data*: these are subjects who have stopped doing the experiment before the measured event has happened to them. This can be either due to the experimenters no longer following the subject (as in the case of the subject exceeding a time limit), the subject withdrawing from the experiment, or for outside effects affecting the subject (for example a subject dying in a bus crash would have to be censored from a study of leukaemia mortality rates).

The Kaplan-Meier method is a way of calculating what is known as the survival function. The survival function reports the probability that for a single subject at a given time the measured event has not yet happened to them. In this regard the Kaplan-Meier method is an extension of life table methods. A life table is one in which subjects are grouped together (classically by age) and each group is given a probability of the measured event happening before the subject advances to the next band. Like the chi-square approach, life tables are sensitive to how the subjects are grouped. The Kaplan-Meier is superior as it does not rely on how the data is quantised. The data is plotted in what is called a Kaplan-Meier survival curve. The curve is a step graph that shows what percentage of the (measurable) population is still surviving at a given point in time. Plotting curves for two (or more) groups gives the experimenter a visual representation of the differences in the survival function for two groups. Normally, as a study progresses and and subjects drop out from the population (are censored), this reduces the total number of subjects and so each event then recorded represent a bigger percentage of the population. This means the graphs have to be read with some care and with reference to the statistical tests that are based on them.

The most commonly used method to then test whether there is a significant difference between two Kaplan-Meier survival curves is the log-rank test. Fundamentally, the log-rank test determines how many events should happen in any given period of time for each group if the null hypothesis is true (that is, there is no significant difference between the groups). Then, the theoretical and actual numbers are compared against each other using

the chi-square test. This test is applied at every observed event time so it does not just test the whole of the population in one calculation. The end result is a standard p-value which is compared against a pre-selected significance level.

Another analysis method for survival curves is called the Cox Proportional Hazard Model [39]. This is a regression model [77] for survival curves. It can deal with both ordered and unordered categorical data as well as continuous and ordinal values. As with all regression models, its purpose is to determine which variables have a significant effect on the time-to-event data and what the relevant importance of those variables are. The Cox Proportional Hazard Model is considered [39] to be a safe choice as, being a non-parametric model, it is very robust and able to cope with data with a variety of theoretical distributions. It is standard to use the Cox Proportional Hazard Model in both uni-variate and multi-variate forms. In the uni-variate approach, a single independent variable at a time is used to see if there is any correlation between it and the dependent variable, providing a standard p-value for determining if the independent variable is a significant contributor to the dependent variable. In this way, those variables that seem to be having an effect on the dependent variable can be identified and examined. The multi-variate approach analysis the effects of all the independent variables simultaneously. The multi-variate approach will end up producing different p-values for each independent variable. This is because in the multi-variate model the interaction of the various independent variables results in different levels of correlation with the dependent variable. Using this approach, relationships between the independent variables can be identified.

An Example

This section consists of an example to show how the correct use of survival analysis affects p-values when applied to experiment data with right-censored data. Table 5.1 shows the results of an experiment to measure how quickly subjects were able to complete a programming task. Half had been working on a object-oriented program and the other half on a procedurally constructed program. They were given 60 minutes to complete the task. Several of the subjects were unable to complete the task in time and as a result were censored from the results at the 60 minute mark.

If the results from these subjects are ignored and the remaining results are used in a t-test (the remaining figures being normally distributed, equally variable and independent) the results give $t = 2.48$ which results in $p = 0.022$. This is beneath the standard $p = 0.05$ level and would be considered a significant result. However, there is no reason to exclude the subjects who did not finish. When a p-value is computed using survival analysis, which uses all the results, the final result is $p = 0.632$. This is a large change in the p value, from being a significant result to being a reasonably non significant result. By examining the survival curves in figure 5.6, it can be seen that although the object-oriented group has an initial advantage once the majority of students have finished there is no real difference between them. This example clearly shows the importance of using the correct statistical tests for performing analysis of quantitative experiment results. The full stats can be found in appendix B.

Subject #	Object-Oriented	Subject #	Procedural
1	11	16	23
2	15	17	27
3	12	18	40
4	37	19	33
5	43	20	52
6	12	21	39
7	39	22	45
8	31	23	34
9	35	24	35
10	22	25	31
11	(60)DNF	26	36
12	(60)DNF	27	(60)DNF
13	(60)DNF	28	56
14	(60)DNF	29	31
15	(60)DNF	30	(60)DNF

Table 5.1: Task Completion

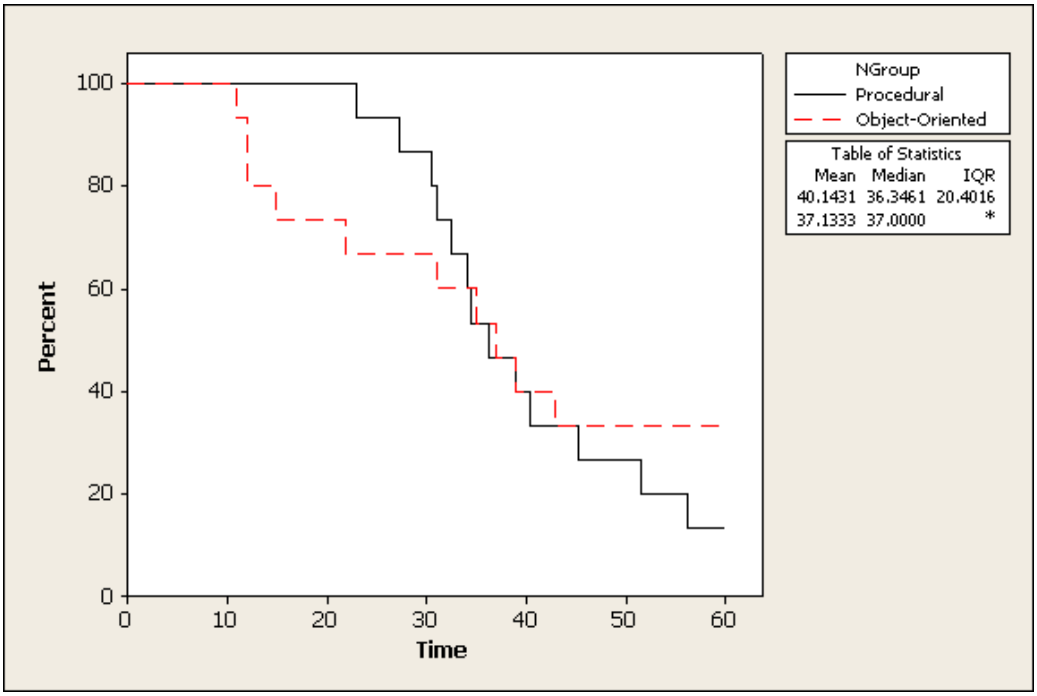

Figure 5.6: Survival Curves by Group

5.9 Conclusion

The various aspects of what needs to be considered when constructing a controlled laboratory experiment have been discussed and considered with particular attention being paid to the appropriateness of the statistical tests being used to analyse the results. While the various components of the experiment have been identified in general, this does not eliminate the need for careful piloting of potential experiment designs. Only by piloting an experiment can specific issues relevant only to a particular design come to light.

However, with the solid foundation this chapter provides I can be confident that the decisions I have made are consistent with established experimental literature in the Software Engineering field.

Chapter 6

Experiment

6.1 Introduction

This chapter describes the design, implementation and execution of an experiment intended to measure the effects on the level-of-understanding of programmers undertaking different maintenance tasks. The design considerations of the experiment are presented with reference to both the fundamentals of experiment design presented in chapter 5 and the unique problems that this experiment presented. The results of the experiment are analysed both quantitatively, with the appropriate statistics, and qualitatively, with reference to the final programs the subjects produced.

6.2 Motivation

Large software systems are usually maintained by teams of maintenance programmers. These teams change over time with members leaving (due to a new job, promotion, reassignment or retirement) and software immigrants joining either as direct replacements for departed members or because the workload of a group has increased. Before software immigrants can become fully productive team members, they have to learn about the software system they will have to maintain. Despite being part of a team, software maintainers tend to specialise on specific sub-systems (see section 3.4.4), so when the immigrant is being brought in to replace a departed team member, they are required to also replace that member's specific knowledge.

Companies often give insufficient thought to succession management [74, 54] and if the outgoing team member is not available, the software immigrant is on their own in terms of learning about the specifics of the sub-system that they are responsible for. Reading system documentation is problematic as documentation is unreliable [64] and it may well be that no-one else within the team has touched the code in months, possibly years. In this situation the only source of information on how the code currently operates is the code itself, and the only way to develop a level-of-understanding about the code is to work with it.

In this context, the experiment was designed to examine how performing different tasks on an unfamiliar piece of code affects subjects' level-of-understanding of that code.

6.3 Similar Experiments

There are a large number of related experiments measuring level-of-understanding effects. They can be broken down into the following categories:

1. Those that examine how the structure of a program affects the ability of subjects to build a level-of-understanding of it [17, 52].
2. Those that examine how varying types of program documentation affects the subjects' level-of-understanding of the program [56, 62, 1].
3. Those that examine how using a particular tool, technique or process aids building a level-of-understanding of the program [75, 71].

While the extensive experimental works of von Mayrhauser and Vans [81, 83, 82] are superficially structured similarly to the experiment design laid out in this chapter, they are interested in program understanding issues rather than level-of-understanding issues, to which they give only the most cursory and non-statistical examination. That is, their experiments are focused around analysing what the programmer does and thinks *while* performing the work whereas my own experiment is focused around working out how much the programmer knows *after* they have done the task.

6.4 Experimental Construction

6.4.1 Purpose of the Experiment

To *measure* the difference in *level-of-understanding* between *programmers* who have performed either *passive* or *active* tasks on an unfamiliar piece of code.

Null Hypothesis 0 – There is no difference in the level-of-understanding of subjects performing either active or passive tasks.

6.4.2 Glossary

Initial task — Any of the separate activities a subject might perform in the first half of the experiment. Each treatment will consist of a different Initial task

Measured task — The activity chosen to be used as the metric of a subject's level-of-understanding.

6.4.3 Constraints

Subjects

The greatest supply of potential subjects was undergraduate computing science students at the University of Glasgow. As this is an experiment looking to examine programming

in general, as high a level of programming experience as possible was desired. This meant that final year undergraduates were selected. They will have had three years of education, and probably some work experience. However, as section 5.3.1 states, there are a number of restrictions they impose which are considered below. Postgraduate students were also available as potential subjects. However, piloting of the experiment showed that they had a highly variable level of ability with insufficient indicators to allow the level of ability to be controlled to avoid biasing the groups.

Time

Given that student subjects are being used, this also constrains the method of inducement (see section 5.3.2). Due to practical financial constraints, a maximum of £20 was available per subject. With normal experimental payments in the university being around £5-£10 an hour this placed an absolute maximum time of four hours for the experiment. Another consideration when using student subjects was the fact that they would have to be taking time out from their studies, and as the identified group was final year undergraduates these would be studies directly impacting on their final degree grades. As a result, the experiment length would need to be minimised in order to make the experiment palatable for the target subject group.

Consultation with a psychologist suggested an upper limit of two and a half hours for the experiment, the principal reason being that he found it difficult to attract undergraduate subjects for experiments that were longer than that. This places a limit of one hour on any potential Initial or Measured tasks, with half an hour for other activity. This time restriction not only affects the nature of the possible tasks, but also the size of the code used in the experiment. As stated in section 5.6, a one hour task length would limit the code used to around 1500 lines.

6.4.4 Basic Initial Task Requirements

Unlike the Measured task, the Initial tasks do not need to provide any form of quantitative results. This is beneficial to the overall design, as it allows the freedom to set tasks that would prove difficult, if not impossible, to measure objectively.

Furthermore, it should take longer than the hour to perform because during the experiment the subjects should be performing the task for the full hour. The experiment is not measuring the effects of *completing*, it is measuring the effect of performing the activities the task requires. As ever though, the task should be of a low domain complexity. This results in a task that will require a large amount of not hugely challenging work.

The Initial tasks should cover the the major parts of the system. If a task explicitly doesn't cover a major component, or more pertinently a component that is necessary to complete the Measured task, then subjects who perform that Initial task are at a disadvantage.

6.4.5 Passive Task

In the context of this experiment, a passive task is one where the subject does not alter or change the code in any way. The passive task represents a view that software immigrants should take a step back from the code and try to learn about it without plunging in and potentially doing more harm than good, both to the code and to their own level-of-understanding of the code.

Reading

The most passive of tasks involving the code would be simply reading it. A possible Initial task would therefore be for the subjects to read the code for one hour before undertaking the Measured task. As the subjects were final year undergraduates, I was unsure as to how well they would be able to self direct themselves for a full hour. There could be a chance that the subjects would grow bored, which would be a confounding factor in analysing the experiment. As the purpose of this group was to have subjects who did not alter the code, an alternative task was selected that fulfilled the goal.

Documenting

Documenting code is a task which involves reading the code without altering or changing the code. In the context of possible actions software immigrants might undertake when taking over code, it is a realistic task given the Pigoski and Looney [54] experience, where they made their maintenance programmers document the code to learn about it. Asking subjects to undertake documentation gives them a level of guidance that does not exist when simply asking them to read the code. It gives them a series of short term goals to work towards, which reduces the possibility of growing distracted or bored due to the open ended nature of just reading.

Resolution

Due to the above considerations, it was felt that asking the subjects to Document the code was an appropriate passive task.

6.4.6 Active Task

In the context of this experiment an active task is one that involves the subjects altering and executing the code. It represents the view that the best way to learn to swim is by getting wet and that more is learnt by doing. Given that the purpose of the experiment is to determine which tasks software immigrants should perform when joining a maintenance team, it follows that the active task should be a maintenance activity.

Various options were considered for the active task. At the basic level there were four types of tasks possible for the subjects to perform: corrective, perfective, adaptive and preventative. The following sections describe the issues surrounding implementing one of these types of tasks as an Initial task.

Adaptive

Adaptive maintenance covers changes to the environment the software operates in. If these changes are due to hardware or operating system changes, the knowledge required to effect those changes will in large part revolve around knowledge of the hardware and operating systems change itself, rather than knowledge of the program. This is an example where varying amounts of domain knowledge would have a large impact on performance. Given that low domain complexity and the reduction of the importance of domain knowledge are key elements to general level-of-understanding experiments, it seems unwise to use such a task.

Other types of adaptive work, mandatory changes to input or output formats, with the attendant changes to processing, results in work that is not greatly different from a typical perfective task. Although the way two tasks were generated, and the priority with which they must be completed could be completely different from the point-of-view of the managers, from the subject's point of view there would be no difference between such an adaptive task and a related perfective task.

Corrective

The use of corrective maintenance (debugging) as a potential Initial task creates two major problems.

The first is to do with the properties of the bugs. The bugs have to be non-trivial so that they cannot be instantly found, however they cannot be so serious as to cause the program to fail instantly. As the program has to be relatively small (1500 lines), this is a difficult sweet spot to hit. Another issue is that if these bugs are introduced *after* the program is created then they are artificial bugs, and there is no evidence to show that artificial bugs match the characteristics of genuine mistakes by programmers.

The second problem comes down to the issue of comparability of materials (see section 5.6.2). There are two options for creating the program to include a debugging task. Either a single program with errors is created which is given to all groups, or two versions of the code are produced, one with errors and one without errors. If a single program is produced, then the expected behaviour of subjects *not* in the debugging group must be considered. Given that they have a specification of how the program should operate, if they spot the anomalous behaviour, or work it out from first principles from the code, they may act differently to how they otherwise would. They may consider that their knowledge of the code is wrong, and make false assumptions based on that. They may "work through" the incorrect behaviour trying to discover what is happening, thus performing similar (although not identical) steps to the subjects in the Debug group. This would blur what effects could be ascribed to being in the Debug group.

If the alternative approach is taken, and multiple versions of the program are constructed the experiment is then, as described in section 5.4.2, no longer comparing the differences between the two treatments that are varying by Process, but also the change in Product as

well. Although it could be argued that the change is minimal in comparison to the changes to programs in the Inheritance hierarchy studies or the Rigi/SHriMP experiments, the arguments for comparability would still have to be made and justified.

Finally, even if multiple versions of the code are successfully argued as being comparable enough to use, then it raises difficulties for the Measured task. Two options would have to be considered. The subjects could work on the different versions of the code depending on what Initial task they performed, with the Measured task being formulated so it was not affected by the bugs in the code, which would complicate the creation of the Measured task. Alternatively, the Debuggers would have to be given the fully working version of the code for the Measured task. This would place the Debuggers at a disadvantage, as they had built up a level-of-understanding about a different program, no matter how comparable it was.

Perfective

Perfective maintenance covers the addition of new functionality to a system. It is probably the single most common programming activity undertaken during maintenance. It is relatively easy to control the size of a perfective task. By adding or removing requirements the task can be made as large as necessary so that it takes longer than an hour to complete without overburdening it with complex domain issues. Similarly, it is easy, when compared to adaptive, perfective and corrective tasks, to make sure the perfective task touches on all the major components making up the program. All of these features combine together to make a perfective task a very desirable activity to use as the active Initial task.

Preventative

Preventative maintenance, covering as it does updating system documentation and re-coding for computational efficiency, seems a poor choice as an active task. Both of these imply a fair working knowledge of the system, which subjects who are fresh to the system will not have. Updating system documentation is partially covered by the passive task that has been selected. Unless the preventative maintenance requires a significant redesign, the type of work that it involves is principally algorithmic change. That is, the type of knowledge gained is about small localised sections of the code, rather than knowledge about the system as a whole. While it would be expected that a subject would have to develop *some* level-of-understanding of the overall system, it was considered that compared to perfective maintenance subjects could "short-circuit" a lot of "necessary" knowledge in order to complete preventative maintenance tasks.

Resolution

The difficulties of using corrective and preventative maintenance tasks were considered to great to use them as an Initial task. Furthermore, there is no discernible difference between an adaptive maintenance task that is appropriate to use as an Initial task and a perfective maintenance task. As a result it decided that a perfective maintenance task was the best choice as the active task for the experiment.

6.4.7 Measuring Level-Of-Understanding

As stated in section 5.5, there are multiple ways of measuring a subject's level-of-understanding. Using a subjective rating or asking the subjects to explain parts of the system were rejected due to the difficulty of producing quantitative results from them. Despite having identified a number of drawbacks with test taking the use of a test was considered in more detail. Whilst the preferred option, as stated in section 5.5, is the use of a maintenance task, it was felt that some attempt should be made to examine other approaches in reference to this experiment. Despite having potential drawbacks, a set of questions was piloted. As well as having the expected problems already identified in section 5.5, it was also found to be difficult to construct non-trivial questions for the program given the size of code. Due to these extra considerations, it was decided that only the results of performing a maintenance task was to be used as the measure of the subjects' level-of-understanding. As half of the subjects will be performing a maintenance task as the Initial task, this introduces special considerations. Fundamentally, the Measured task cannot be too similar to the Initial maintenance task. If it is, then the subjects performing the Initial maintenance task are receiving training in succeeding at the Measured task.

Desirable Properties of the Measured Task

The most desirable property of the Measured task is that a subject with a high level-of-understanding of the experiment code would be able to complete the Measured task as fast as they can write code. As is consistent with the conclusions of section 5.6, this means that the Measured task should be of a low domain complexity and that the implementation of the task should require no complex programming. If the domain of the task is complex then the subject may be confounded by a lack of knowledge of the domain, despite having a high level-of-understanding of the code. If the coding required is complex then the subject might be confounded despite knowing, in theory, what it is that must be done. Counter balancing this is the problem of ceiling effects, as the task cannot be something so trivial that a subject with comparatively low level-of-understanding can complete the task as fast as a subject who has developed a high level-of-understanding. At the other extreme, although there will always be subjects who will not be able to complete the Measured task, it is imperative that the number of failures is kept to a minimum. If too many subjects fail to complete, it will be difficult for the statistical tests to produce reliable results. These competing demands for the qualities of the Measured task mean that piloting is the only worthwhile approach to determining if a developed task is appropriate for the experiment.

The same considerations that guided the selection of the Initial maintenance task still hold with selecting the Measured task. As a result, once again a Perfective task was selected. Producing documentation, being a form of Preventative maintenance, was given consideration, but ultimately it is another way of explaining how the system operates, which has already been rejected as being too subjective.

Perfective maintenance can be roughly split into two different types: work that adds additional functionality to the system (additive), and work that changes how a currently implemented feature operates (changeative). To help avoid the possibility of the Initial

task explicitly training for the Measured task, it would be preferable that the two tasks be of different types. It is probably inadvisable to make the changeative task the Initial task, as subjects would be working on a fresh copy of the program for the Measured task. This would mean that those subjects who had made changes to the code would have been building a level-of-understanding about the modified program rather than the fresh one used for the Measured task. If the Measured task made use of any of the sections that were heavily changed, the subjects may well get confused as they conflate their changes with the initial system state. Having the chageative task as the Measured task eliminates this possibility so is obviously preferable.

Quantitative and Qualitative Measures

While chapter 5 emphasises quantitative methods and measures that produce quantitative results that does not mean that qualitative results are simply ignored: they form a valuable component of analysing the results. However, every effort has been made to remove subjective judgement from the data used in the quantitative analysis. The Initial tasks chosen will each result in materials being produced. These materials can be examined to look for commonalities and differences as well as being judged for subjective quality. This information can be used to qualify the quantitative results.

6.5 Experiment Instantiation

This section describes an experiment that fulfills the goal of section 6.4.1 while taking in the considerations of sections 6.4.3-6.4.7.

6.5.1 Basic Overview

Subjects were split into two groups. Both groups undertook an Initial task for one hour. The first group Enhanced the code while the other group Documented the code. Then, the subjects were timed while adding a new feature to the code, and their time taken to complete this Measured task was used to determine their level-of-understanding of the code.

6.5.2 Hypothesis

Null Hypothesis 1: There is no difference in the level-of-understanding of the Enhancement and Document groups.
Hypothesis 1: There is a significant difference in the level-of-understanding between the Document and Enhancement groups.

6.5.3 Subjects

Subjects were 4th year Computing Science students at the University of Glasgow. This meant that each subject had three years' programming education with at least one year of programming experience in Java and a grade in a Java based programming module. Subjects were offered £20 to participate in the experiment.

6.5.4 Procedure and Measures

The subjects were split into two groups using stratified random sampling, using their programming grade and a subjective self-rating as a Java programmer on a scale of 1 to 10. Their grade was the major component used to stratify them, with the self-rating used to split up subjects with the same grade. The two groups were labelled Enhance and Document. The subjects were given a demonstration of the system that they were to be performing their tasks upon. They were then given as much time as necessary to read over a written specification of the system and to ask any questions they had. Subjects were then given one hour to perform the relevant Initial task (either Enhancement or Documentation) on the system. The subjects were then given a ten minute break in which snacks and drinks were given to them, and they were engaged in discussion about topics other than the experiment. Then, working from a fresh version of the system, subjects in both groups undertook the same second (Measured) task which was an enhancement task. The subjects were given at most one hour to finish the second task, and the length of time taken to successfully complete it was used as the metric of their level-of-understanding of the code. Subjects were then debriefed and asked about their general understanding of the system and approach to tackling the task.

Subjects worked within the standard Linux environment using their preferred text editor and any command line tools they felt were appropriate, but without using IDEs. Subjects were allowed to access the Java SDK web pages but no other websites were allowed to be accessed.

6.5.5 Materials

A single Java program, details of which are provided below, was used as the experiment code. A written specification of the system was provided along with a basic class diagram of the code. An example piece of documentation was provided to show the Document group what level of detail was desired. A specification of an enhancement task for the Initial Enhancement group and an enhancement specification to act as the Measured task were also provided. Finally, a written description of the three tasks was also required. A blank sheet of paper was provided for note taking. All experiment materials are shown in appendix C.

6.6 Experiment Details

Despite being based on the solid grounding of reading of established experimental literature, every experiment has its own unique set of problems caused by the exact combination of tasks, structure and intent of the experiment. This section examines the issues involved in constructing and running the final experiment design. These issues were identified by a combination of the results of piloting various forms of the experiment design, and insight into the unique issues that this level-of-understanding experiment has. Many of the issues were raised by the piloting of an experiment design described in appendix D. The Initial and Measured tasks are discussed, explaining why they were selected and alternate tasks

rejected. The nature of the experiment code is described to justify its creation and aid discussion of the qualitative results in section 6.8.4.

6.6.1 Design Fundamentals

The experiment involves two treatments, one Documenting the code, the other Enhancing the code. This fulfills the need for groups to perform passive and active tasks. The groups' level-of-understanding is measured through performing a second enhancement task on the code. This is a Type one experiment as described in section 5.2, as the change in task is a change in Process whilst the People and Product remain the same in each treatment.

As it is a between-groups design, attention must be paid to balancing groups for external factors that can affect performance in the experiment. It was considered that the only measurable factor that would have an effect was general programming ability. Given that the purpose of the assessment of the Java programming course was to determine the subject's ability as a programmer, it was felt that the use of their grade was a justified mechanism to stratify the subject population before assigning them to groups.

Fatigue

Subjects were given a ten minute break in the middle of the experiment. This was to try and minimise any fatigue effects (see section 5.4.4) that might occur due to the relative long length of the experiment, even though fatigue effects are less relevant to between-groups experiments. Another reason was to avoid the possibility of subjects needing a bathroom break during the experiment tasks, which would have to result in the subject's time being a censored measurement at the time they took the break.

6.6.2 Constructed Program

Real vs Constructed Program

As stated in section 5.6 it is preferable to use a real-world program rather than creating a program specifically for the experiment to avoid problems of potential bias. The pilot was run using a real-world piece of code. While it was of the desired size for the experiment, approximately 1500 lines of code, it had two properties that caused it to fail by the guidelines set out in section 5.6.

The major problem was that it used the Swing programming library. This made a reasonable knowledge of the basic functions of the library a prerequisite to performing the experiment. While the potential subjects, 4th year Glasgow University computing science undergraduates, were guaranteed to have taken at least one course that covered using the library, it was discovered that many did not like using the library and as a result were put off by having to work with Swing even though they had the level of knowledge required.

The second problem was that the program had quite a high level of domain knowledge attached to it. The program was a zoomable graph display with a pair of inbuilt clustering

algorithms. This meant that there was a large number of mathematical formulae in the code, which, while not overly complex, would have given the subjects with a good grasp of trigonometry a definite advantage over mathematically weaker subjects.

Experiment Code

Due to the problems of finding a a piece of code that could fulfil the requirements for the experiment, it was decided that a piece of code would be constructed specifically for it. While constructing code specifically for an experiment can raise issues of the construction biasing the result, this experiment is exploratory rather than trying to validate a specific Process and as a result, this greatly diminishes the possibility of deliberate bias. The full specification and listing of the code is given in appendix C.9 but an overview is given here to aid discussion of the results.

The program was a simple command-line interface with commands which would create and manipulate sports ranking systems. There are two domains involved: the command-line shell and the nature of the sports ranking systems. The subjects, being computing science students with at least one year's experience using Linux, would be familiar with command line paradigms. Similarly, the fundamentals of the sport ranking systems are very simple, with no great depth to understanding how they work. Furthermore, an extensive description of all three ranking systems was given prior to the subjects undertaking the tasks. As such, there are no problems of domain complexity confounding the results of the experiment.

The program itself consists of two main sections: the code for the command line interface (the parsing, command generation and command objects) and the ranking system implementations, of which there were three, all presented to the rest of the system through an interface. The addition of a new command to the system (which both the the Initial Enhancement task and Measured task require) requires knowledge of both of these parts of the system.

The operation of the system follows this format: an input line is passed into the currently loaded command factory (the appropriate command factory is instantiated depending on which of the three different ranking systems is loaded). The factory checks to see if it recognises the first token of the line as a command, if not it passes it up the inheritance hierarchy until one of the super-classes does. Once the command is recognised a `command` object is instantiated and the arguments for the command (if any) are passed in along with a reference to the object representing the currently loaded ranking system. The command then calls the necessary code in the ranking system to perform its function and then formats and passes back the output of the command to the command line interface which then displays it.

The constructed program fulfilled the requirements laid out in section 5.6. It is of an appropriate size, approximately 1500 lines, to be used in an experiment involving two one-hour tasks. It has a known, and low, domain complexity. Finally, the code had low

implementation complexity with a degree of modularity thus avoiding confusing the subjects with obtuse programming tricks.

The program was written in Java because that is what the subjects had been most recently taught and had extensively used over the previous year. It was therefore chosen due to expediency, rather than because it was thought to have any special properties that were beneficial to the experiment. Conversely, the use of Java was not thought to have any negative properties that would adversely affect the experiment.

6.6.3 Initial Enhancement Task

This task requires the addition of a single level undo facility to the system. As this involves adding a command to the system, it requires the subjects to know how Command objects are generated in the system as well as requiring a reasonably high level-of-understanding of the internals on the three ranking systems, and knowing how the ranking systems' internal data structures work. Whilst there is only one way of generating a correct Command object, there are multiple ways in which the undo functionality can be implemented, with varying levels of sophistication.

The key proposition is that the undo command is a relatively challenging task, considering the ability level of the subjects, so would be unlikely to be completed in the hour. Furthermore it covers the key parts of the system: command generation, execution and interaction with the ranking systems. In that way it meets the required properties of section 6.4.6

6.6.4 Documentation and Subjective Value

Due to the code being in Java, the Documentation the subjects were asked to produce was JavaDoc. This style of documentation was chosen as it is a style the subjects are familiar with, and it does not require the creation of extra files, thus making the gathering and analysis of the documentation easier. Requiring the subjects to produce other forms of documentation was considered but rejected: subjects were provided with a basic class diagram so there was no scope for asking them to produce such a document. The subjects already had a specification of the code so there was no need for them to reverse engineer. This left adding comments to the code in the form of JavaDoc as the only small scale approach to documenting the program in a structured way.

In the pilot of the alternative design, subjects produced program documentation much as they do in the final design. However, this documentation was awarded a score and this score was used as a measure of the subjects' level-of-understanding. As described in section 6.4.7, the use of subjective measures for quantitative results has been avoided in the final experiment design. This was partly informed by the difficulty in confidently rating the quality of subjects' documentation in the pilot experiment. Furthermore, producing good code-level comments is not a skill which is practised or examined by the University of Glasgow's Computing Science undergraduate course. As a result, even if the documentation was able to be accurately judged, using the documentation produced

as an indicator of a subject's level-of-understanding would still be a confounding factor in the experiment design given that it is impossible to balance the subjects for their ability to produce documentation. As a result, the final experiment design was influenced by the desire to avoid the need to perform quantitative analysis of the subjects' documentation for the purposes of performing the statistical tests. This aim was achieved, and while the documentation the subjects produced was read and analysed it was purely for a qualitative view.

6.6.5 Measured Task

Problems in Creation of Task

The final experiment design was piloted with three subjects. None of them could complete the Measured task (described in appendix C.7). Not only that, but in their debriefing they stated that they had no idea how to go about undertaking the Measured task. This was potentially extremely damaging because if these subjects were representative of other potential subjects, then no subjects would be able to finish the Measured task and the experiment would collapse, as there would be nothing to measure.

There were four possibilities as to why the subjects were unable to finish the Measured task: the subjects could not gain a great enough level-of-understanding about the system to complete the task; the task was too programmatically complex to implement in the available time; the task was too domain complex for the subjects to understand; or the subjects were not good enough programmers overall. In questioning the pilot subjects, it was determined that the task was too *domain complex*: the subjects were able to explain what the various parts of the system did in a cohesive manner but they did not grasp what was totally required of them for the Measured task.

Two options were considered for correcting this problem: additional explanation of the Measured task or a new Measured task that had lower domain complexity. There were time constraints to consider, in that the subjects were only available for two weeks before their course-work started to mount up and they became unavailable to do the experiment. It was felt on balance that the current Measured task was going to be too domain complex even after providing additional explanation and that the risk could not be taken as even more time would be lost if the extra explanation failed. As a result, it was felt that producing a new Measured task was the safest approach to take. The Measured task designed (adding the functionality to query the existence of a specific player name in the currently loaded ranking system, described in appendix C.5)had a considerably lower domain complexity but was still spread across the whole system, thus requiring a wide range of knowledge about the system to complete. The Initial Enhancement task was also slightly modified to reduce the similarity between it and the Measured task. This new task does have a greater similarity in characteristics to the Initial task than is desirable; they are both additive Enhancement tasks which require adding a new command to the system. Attempts were made to create a new chageative task, however, the two chageative tasks produced (described in appendix C.8) were considered to be too trivial and too localised, thus risking the introduction of a ceiling effect. As practising Enhancement is not

practising any specific skill other than programming, which is something the subjects are supposed to be able to do, it was felt that the additive Enhancement task was not going to bias the results of the experiment.

Completion of Task

The subjects were given a written specification of the Measured task which contained a description of what was required for the tasks to be considered successful. A series of test commands were constructed to determine if the Measured task was completed successfully. This tested the successful running of the new command on the three different ranking systems as well as the error handling capabilities of the code. The tests were made available to the subjects so that they could measure their own progress while they attempted the Measured task.

Using a Test

The use of some form of test was considered. Whilst the difficulties of using them as the *sole* measure of a subject's level-of-understanding was all ready recognised in section 5.5 and section 6.4.7, it was thought that a test could be administered *in addition* to the use of a maintenance task as the primary measure of the level-of-understanding to act as a verification measure.

The use of a test in such a role was included the the pilot of the alternative design. This revealed that in practice the answering of the questions raised an interesting point about the accuracy of the answers. One subject answered the questions in far more detail than was necessary to score full marks by the given marking scheme. This meant that he answered a smaller number of questions than other subjects despite the fact that, as was clear from from debriefing and the detailed answers, he had a greater level-of-understanding about the code than all the other subjects who took part in the pilot. An example of the type of depth he went into is in his response to the question *"What is the minimum time the zoom animation can take to run?"*. There was a constant timing variable of 700 milliseconds, which most subjects found and reported. However, this subject instrumented the code and collected runtime data for both zoom-in and zoom-out operations and gave average and lowest-measured timings for both.

A second issue for the test relates to where a test or tests should be inserted in the final experiment design. If it is placed *before* the Measured task then the experiment no longer measures the difference between Enhancing and Documenting but between Enhancing & test taking and Documenting & test taking, as discovering answers to the test questions is a form of learning and thus would affect a subject's level-of-understanding. If the tests are taken *after* the Measured task has been performed then the issue of a subject's total time with the program becomes problematic. A subject who completes the Measured task in ten minutes would be judged to have a greater level-of-understanding than a subject who finished in 50 minutes. However, they will only have had 70 minutes of time in total with the code while the other subject will have had 110 minutes. Whilst the results of the test

may be very interesting, it makes the test results much more problematic as a measure of a subject's overall level-of-understanding which is the *core* purpose of the experimental design.

Due to these issues it was decided that there would be no test of any form in the final experiment design as the complications and difficulties it presented, along with the increase in experiment length it entailed, were considered too great when compared against the minimal benefits.

6.6.6 No Integrated Development Environments

There were two reasons that the subjects were restricted to using text editors rather than an Integrated Development Environment (IDE). The first is the additional utility that the IDE can offer: a proficient user of an IDE would have a large, but unquantifiable advantage over a subject not using an IDE. While an experiment measuring how IDEs affect a subject's level-of-understanding of a system would be interesting and useful, that is not the purpose of this experiment. The second reason is setup time: many IDEs require setup time for importing new code, and although this does not take long in relation to a normal project's lifecycle, even ten minutes' setup time would be a significant amount of time not spent on the experiment. Whilst this restriction on IDE usage marginally reduces the realism of the experiment, the increased homogeneity of the subjects was judged more important.

The subjects were, however, allowed to use any of the standard **Unix** command line tools that they felt were appropriate. It could be argued that these tools would have the same effects as the the features in an IDE. It was felt that it was appropriate to let the subjects use these tools as they are an integral part of developing software in the **Unix** environment, and that to deny them access to **grep** and the like would be to limit their productiveness too much. Furthermore, the subjects all have a similar level of **Linux** experience. In the end, I discovered from debriefing the subjects after the end of the experiment that no command line tools other than **grep** had been used. Furthermore, even **grep** was only used to a minimal extent and by only a few of the subjects.

6.6.7 Provision of Class Diagram

A basic class diagram was provided to the subjects, which showed only class inheritance hierarchies and interface implementation. It was thought that it would be a significant time investment for the subjects to produce such a diagram for themselves and a basic class diagram is almost a prerequisite of learning about an OO system. Given the prevalence of tools that can automatically recover a system's class diagram, to varying levels of detail, it was thought that it was only reasonable to provide subjects with a basic class diagram.

6.7 Running the Experiment

The experiment was run on three separate occasions, Autumn 2003 (cohort 1), Autumn 2004 (cohort 2) and Autumn 2006 (cohort 3). The experiment was run in Autumn as it

was considered the best time to be able to attract potential subjects. Running at any other time of year would have clashed with practical assignments, holidays or exams. This was shown by the attempt to get participants for the alternative design pilot, which was run at spring time and received a very low response rate from undergraduate students. The experiment was run on all days of the working week. It was run at the end of the academic day, 4:30pm, to avoid clashing with subjects' lectures, and also to minimise any potential fatigue effects. To speed up the overall running of the experiment subjects were run through in small groups: up to four people performed the experiment simultaneously, although there were also executions with only a single subject. Subjects were forbidden to communicate with each other during the experiment, except during the ten minutes break where conversation was kept off the system they had been working on.

6.8 Results

6.8.1 Reasons for Repeating the Experiment

As stated the experiment was run three times, in three different years. The reason for this was that the first time the experiment was run there was no significant result for the main hypothesis. However, as can be seen from tables 6.1 & 6.2, only six of the ten Enhancers from the first run of the experiment (cohort) finished the Measured task (and all in 30 minutes or less) while nine out of ten of the Documenters completed the task. This suggested a certain bi-modality in the Enhancers: that if they were good enough, Enhancing was the best way to learn. On the other hand the more high level view that Documenters took, while not imparting as much information to the subjects, gave them enough of a overview to allow them to find the information they needed to complete the Measured task.

As the number of participants (20) was somewhat low to produce reliable results, the decision was made to run the experiment again with a further 18 subjects (2 Enhancers pulled out at the last minute). As can be seen from table 6.3, the 2nd cohort Enhancers seem to be of a different character to the 1st cohort Enhancers, with much higher mean and median times to completion. However, the survival analysis shows no significant difference. Furthermore, there seemed to be no change in the Documenter groups. The manner in which the experiment was run was reviewed to try and identify any factors that may have sped up or slowed down the two Enhancement groups. No variation was found in how the experiment was run, so external factors were also examined. An examination was made of the way that the programming courses had been taught as they might have changed between the years. A more detailed examination of the subjects' academic results was also undertaken in case the grades where masking high/low variations in the quality of the As and Bs. Once again, no differences of any note were found: the programming courses were run using the same material by the same lecturers as they had been the previous year, and the more detailed grade analysis gave no further insight. As a result, the experiment was run a third time to try and find if there was a trend toward bi-modality of results in the Enhancers. On this occasion only 15 subjects could be attracted to perform the experiment, and due to the desire to primarily examine differences in the Enhancement group, it was run this time with ten Enhancers and five Documenters.

Cohort	Sub Num	Initial Task	Time	Grade	Rating
1	5	Enhance	12	A	8
1	6	Enhance	24	C	5
1	11	Enhance	Non–Comp	B	5
1	12	Enhance	Non–Comp	C	6
1	13	Enhance	19	A	9
1	16	Enhance	24	B	7
1	18	Enhance	Non–Comp	A	9
1	19	Enhance	9	A	7
1	21	Enhance	30	B	6
1	23	Enhance	Non–Comp	None	8
2	29	Enhance	Non–Comp	C	7
2	30	Enhance	24	A	7
2	36	Enhance	39	A	6
2	38	Enhance	21	B	8
2	39	Enhance	Non–Comp	C	4
2	41	Enhance	59	C	7
2	42	Enhance	19	A	7
2	44	Enhance	51	A	6
3	52	Enhance	39	B	7
3	55	Enhance	28	A	7
3	56	Enhance	26	A	6
3	58	Enhance	30	C	7
3	59	Enhance	23	A	5
3	60	Enhance	10	A	7
3	62	Enhance	Non–Comp	C	6
3	64	Enhance	52	B	7
3	65	Enhance	41	A	6
3	66	Enhance	16	A	7

Table 6.1: Enhancers' Details

Cohort	Initial Task	Initial Task	Time	Grade	Rating
1	4	Document	31	C	6
1	7	Document	52	A	6
1	9	Document	16	A	8
1	10	Document	58	B	7
1	14	Document	34	A	7
1	15	Document	Non–Comp	C	7
1	17	Document	27	A	8
1	20	Document	47	B	7
1	22	Document	36	B	6
1	24	Document	29	A	9
2	28	Document	25	A	8
2	31	Document	Non–Comp	C	2
2	32	Document	33	A	7
2	33	Document	Non–Comp	B	5
2	35	Document	17	A	5
2	37	Document	19	A	8
2	40	Document	56	B	6
2	43	Document	17	A	7
2	45	Document	43	A	7
2	46	Document	34	A	7
3	51	Document	60	C	7
3	53	Document	Non–Comp	A	6
3	54	Document	52	E	4
3	61	Document	43	B	7
3	63	Document	Non–Comp	A	6

Table 6.2: Documenters' Details

Group	Mean Time To Completion	Median Time To Completion
1st Cohort Enhance	23.8	24
2nd Cohort Enhance	41.4	39
3rd Cohort Enhance	31.7	28
1st Cohort Document	38.8	34
2nd Cohort Document	35.6	33
3rd Cohort Document	55.0	60

Table 6.3: Mean and Median Time to Completion

6.8.2 Hypothesis 1 Result

In comparing the Enhancers with the Documenters using survival analysis the result is $p = 0.717$ as seen in table 6.4. This leaves no significant difference between the Enhancers and Documenters and as such Null Hypothesis 1 cannot be rejected. This means that there is no statistically significant difference in the level-of-understanding gained by subjects who performed Enhancing over subjects who performed Documenting. By the design hypothesis, Hypothesis 0, this means that there is no significant difference between subjects undertaking an active task and subjects undertaking a passive task. The survival curve can be seen in figure 6.1. As can be seen, although in the early period it looks as if the Enhancers will be the faster group after 30 minutes the completion events start to spread out. On the other hand the Documenter completion times seem to be slightly more consistent although starting later than the earliest Enhancers. Overall, despite reasonably different median times for the Enhance and Document groups the curves are not that different, as revealed by the statistics[1].

Variable	Chi Square	DF	p-value
Initial Task	0.131778	1	0.717
Cohort	0.008772	2	0.996
Grade	10.0706	2	0.015
Self–Rating	5.65624	1	0.017

Table 6.4: Log–Rank Tests

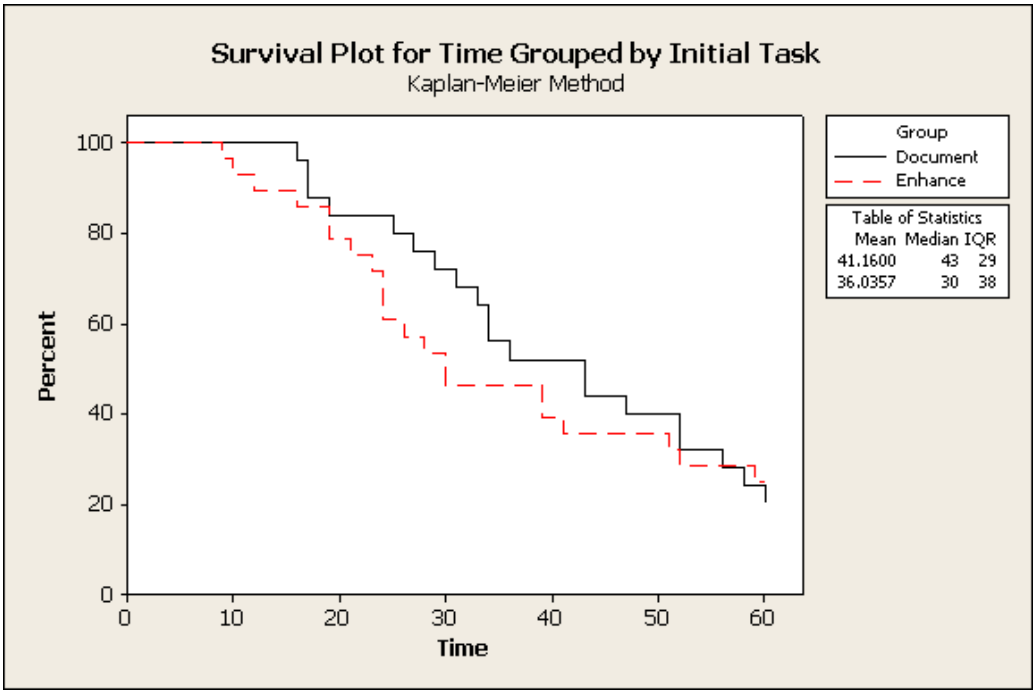

Figure 6.1: Survival Curves by Initial Task

[1] Full statistical working for these and all other statistical test in this chapter can be found in appendix A

6.8.3 Balance

As stated in section 6.6.1, this experiment uses a between-groups design and as such the balance of the groups is important for determining if the results of the experiment are reliable. If one group has a greater proportion of better programmers it would probably be no surprise if that group was faster. At the same time, it would be interesting to note if programming ability, as measured by the subjects' module mark was a good judge of ability. These two ideas can be stated in the form of a hypothesis:

> Null-Hypothesis 2: There is no significant difference in the completion times of subjects with different programming grades.
> Hypothesis 2: Subjects with better grades will have faster finishing times than subjects with lower grades.

A second assumption is that the subjects can accurately self-assess their programming ability. Once again this can be stated in the form of a hypothesis:

> Null-Hypothesis 3: There is no significant difference in the completion times of subjects with differing self-ratings.
> Hypothesis 3: Subjects who have rated themselves with a higher self-rating will have faster completion times than subjects with lower self-ratings.

These two hypothesis were tested using survival analysis. As with main hypothesis, with no compelling reason not to, significance was tested for at the $p = 0.05$ level.

Kaplan-Meier survival curves were computed for the subjects' programming grade, shown in figure 6.3, and analysed with the log-rank test, giving a final result of $p = 0.015$ which is a significant result, allowing the null-hypothesis to be rejected. To analyse the subjects' self-rating using Kaplan-Meier survival curves they were split them into two groups, '6 and below' and '7 and above' seen in figure 6.4. This gave a result 0f $p = 0.017$, once again a significant result, although this hides some strange fluctuations, for instance the subjects who rated themselves an 8 are better than those who rated themselves 6, but not those who rated themselves 5. This does raise some doubt as to the ability of subjects to rate themselves. As a result self-rating was also used as the independent variable in a uni-variate Cox Proportional Hazard Model to determine if it was a significant indicator of the completion time. The resultant p-value was $p = 0.021$ which is a significant result. However, as is standard, all independent variables (cohort, group, grade and self-rating) were analysed in a multi-variate Cox Proportional Hazard Model, which produced the result that *only* grade was a significant indicator of completion time, with self-rating having a result of $p = 0.158$. This suggests that while self-rating was a moderately accurate way of predicting completion times for all subjects it is also strongly correlated with the subjects' grades and that grade is a far more accurate indicator of completion time than self-rating. Thus, once the completion times are balanced and blocked by grade the self-rating is not a particularly useful indicator of ability of subjects within each grade band.

As a subject's grade (and to a lesser extent self-rating) is a statistically significant indicator of ability, it is important to show that the two groups were balanced for ability. Tables 6.5 & 6.6 show the distributions of subjects for each grade and self-rating between the

groups with the theoretical perfect number of subjects per group. As can be seen each distribution is within a single subject. This means that the groups were as well balanced as possible and that they were not biased by ability.

Differences between cohorts was also tested for, but absolutely no significant difference (p=0.996) was detected as is clear from figure 6.2. This strongly suggests that there was no material difference in the subjects between the years the experiment was performed in.

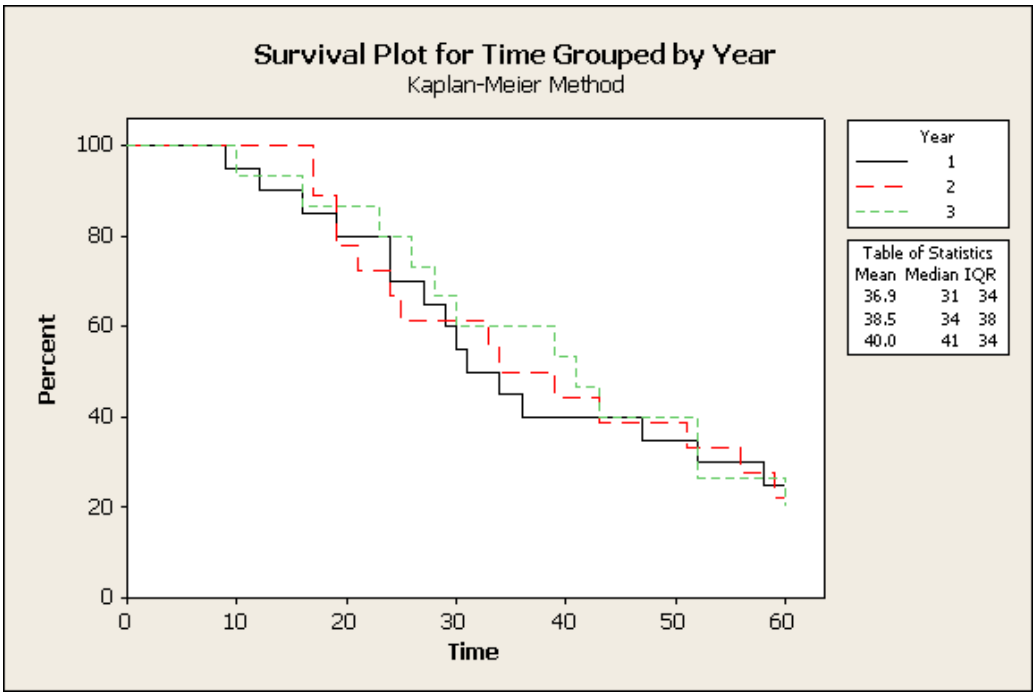

Figure 6.2: Survival Curves by Cohort

Grade	Actual Enhancers	Theoretical Enhancers	Actual Documenters	Theoretical Documenters
A	15	15.3	14	13.6
B	6	6.3	6	5.66
<= C	7	6.3	5	5.66

Table 6.5: Number of Subjects by Grade

Self–Rating	Actual Enhancers	Theoretical Enhancers	Actual Documenters	Theoretical Documenters
9	2	1.5	1	1.4
8	3	3.7	4	3.3
7	12	11.6	10	10.3
6	7	6.8	6	6.1
5	3	2.6	2	2.3
4	1	1	1	0.9
2	0	0.5	1	0.5

Table 6.6: Number of Subjects by Self–Rating

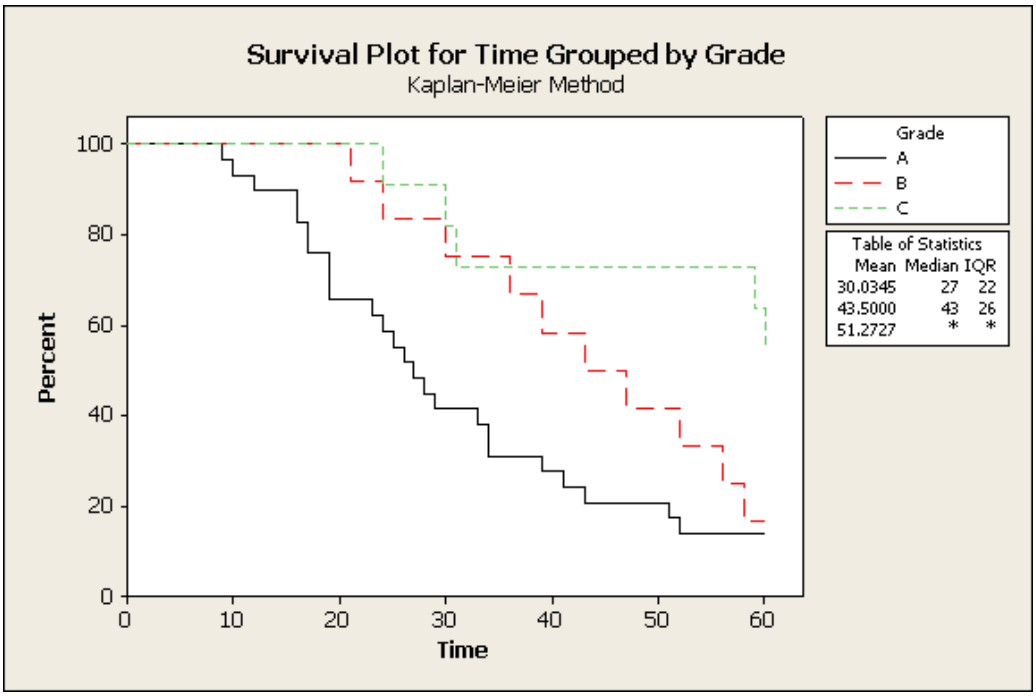

Figure 6.3: Survival Curves by Programming Grade

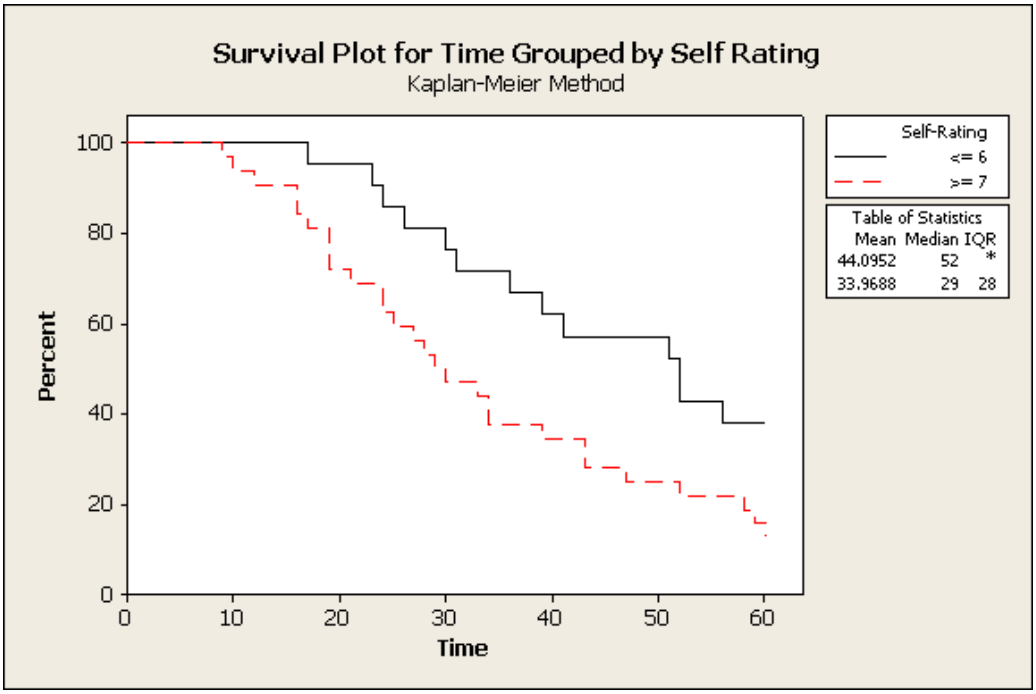

Figure 6.4: Survival Curves by Self-Rating

	4	7	9	10	14	15	17	20	22	24
AddPlayerCommand.java		×	×		×		×			
AlreadyStarted.java		×	×		×			×		×
BasicCommandFactory.java	×	×	×	×	×	×		×	×	×
BasicListCommandFactory.java	×	×	×	×	×	×		×	×	×
BasicResult.java		×	×	×	×	×		×	×	
Command.java	×	×	×	×	×	×	×	×		×
CommandFactory.java	×	×	×	×	×	×		×		
CreateCommand.java		×	×		×		×			
ErrorCommand.java		×	×		×	×	×	×		×
HelpCommand.java		×	×		×	×	×	×		×
HistoryCommand.java		×	×		×	×	×	×		×
IncorrectArguments.java		×	×		×			×		×
LadderSystem.java	×	×	×	×	×	×	×	×	×	×
LeagueSystem.java	×	×	×	×	×	×	×	×	×	×
ListCommand.java		×	×		×	×	×	×		×
LoadCommand.java	×	×	×		×		×	×	×	×
NameAlreadyExists.java		×	×		×			×		×
NameDoesntExist.java		×	×		×			×		
PointsShell.java	×	×	×	×	×		×	×	×	×
QuitCommand.java		×	×		×	×	×	×		×
RankingSystemI.java	×	×	×	×	×	×	×	×	×	×
RankingSystemLoader.java	×		×	×	×			×		×
ResultCommand.java		×	×		×	×	×	×		×
ResultI.java	×	×	×	×	×					
SameNameException.java		×	×		×			×		
SaveCommand.java			×							
Shell.java	×	×	×	×	×	×		×	×	×
SystemAlreadyExistsException.java			×							
UnknownCommand.java		×	×		×			×		×

Table 6.7: First Cohort – Comments Added To Files

	28	31	32	33	35	37	40	43	45	46
AddPlayerCommand.java			×	×		×	×	×		×
AlreadyStarted.java		×	×				×			
BasicCommandFactory.java			×		×	×	×	×	×	×
BasicListCommandFactory.java			×		×	×	×	×	×	×
BasicResult.java	×		×		×		×	×	×	×
Command.java	×		×	×	×	×		×	×	×
CommandFactory.java	×		×		×	×		×	×	×
CreateCommand.java		×	×	×		×	×	×	×	
ErrorCommand.java			×	×		×	×	×		
HelpCommand.java		×	×	×		×	×	×		
HistoryCommand.java		×	×			×	×	×	×	
IncorrectArguments.java							×			
IsInCommand.java		×								
LadderSystem.java	×	×	×			×	×	×	×	×
LeagueSystem.java		×	×			×	×	×		×
ListCommand.java		×	×	×		×	×	×	×	
LoadCommand.java	×	×	×	×		×	×	×		
NameAlreadyExists.java			×				×			
NameDoesntExist.java			×				×		×	
PointsShell.java		×			×	×	×	×		
QuitCommand.java			×	×		×	×	×	×	
RankingSystemI.java	×	×			×	×	×	×	×	×
RankingSystemLoader.java				×					×	
ResultCommand.java				×		×	×		×	
ResultI.java	×		×		×			×		
SameNameException.java			×				×			
SaveCommand.java						×				
Shell.java	×	×	×		×	×	×	×		×
UnknownCommand.java				×			×			

Table 6.8: Second Cohort – Comments Added To Files

	51	61	63	67
AddPlayerCommand.java	×	×	×	
AlreadyStarted.java	×	×	×	
BasicCommandFactory.java	×	×	×	×
BasicListCommandFactory.java	×	×	×	×
BasicResult.java	×	×	×	
Command.java			×	×
CommandFactory.java	×	×	×	×
CreateCommand.java	×	×	×	
ErrorCommand.java	×	×	×	
HelpCommand.java	×	×	×	×
HistoryCommand.java	×	×	×	×
IncorrectArguments.java	×	×	×	
IsInCommand.java				
LadderSystem.java	×	×	×	×
LeagueSystem.java	×	×	×	×
ListCommand.java		×	×	
LoadCommand.java	×	×	×	
NameAlreadyExists.java	×	×		
NameDoesntExist.java	×	×	×	
PointsShell.java	×	×	×	×
QuitCommand.java	×	×	×	
RankingSystemI.java	×	×	×	×
RankingSystemLoader.java	×		×	×
ResultCommand.java		×	×	
ResultI.java	×		×	×
SameNameException.java		×	×	
SaveCommand.java			×	
Shell.java	×	×	×	×
UnknownCommand.java	×	×	×	

Table 6.9: Third Cohort – Comments Added To Files

6.8.4 Qualitative Discussion

Results of Initial Tasks

As expected, none of the Enhancers finished the Initial task in time. There was a very wide distribution of work attempted, ranging from not altering a single file to being a few bug fixes away from completion. Unlike for the Measured task, the subjects who made a significant effort all adopted different approaches. There is a fairly weak link between the amount of work done in the Initial Enhancement task and the subjects' performance in the Measured task: generally the more they had done the better they did, although one of the fastest times was produced by a subject who had barely touched the code in the Initial task.

For the Document task there was once only a weak link between the number and quality of the comments in relation to how well they performed on the Measured task. The comments produced were of highly variable quality, with some subjects producing a large quantity of very poor quality documentation. For example, subject 17, although not commenting on all the key classes, produced vastly superior documentation to subjects 4 and 7. There were eight classes were defined as being 'key' to the operation of the system: as can be seen from tables 6.7, 6.8 and 6.9 the majority of the Documenters commented these classes. Four of the five documenters who failed to complete the Measured task either did not comment the majority of the key classes or produced *very* poor quality documentation of the classes. Subject 35, who completed the Measured task in 17 minutes, the second fastest Documenter time, embodied the difficulty of trying to perform quantitative analysis on documentation. They commented only nine classes in total, only four of which were of the identified "key" classes. However, the comments they did provide were of the highest quality and the most informative of any of the comments. If some form of quantitative marking scheme was used then they would have obtained the maximum possible score for the comment but they would have had a small number of points overall as they would have missed out the large scoring key classes.

Interestingly three subjects documented a class called `SaveCommand.java`, and they gave a full and complete description of its purpose. However, `SaveCommand.java` was a piece of legacy code from a previous version of the program. Nowhere in the program was an instance of *SaveCommand* ever instantiated. Two of the subjects were the 1st and 4th fastest Documenters in the Measured task, whilst the 3rd had a average time for a Documenter.

The variation in documentation quality and quantity is a practical illustration of the difficulty of using a quantitative approach to attempt measure success at documenting. While it would be possible to create a rubric to mark the documentation, the variation in scores and disconnect between those scores and performance at other tasks would make such marks a very poor quantitative measure of a subject's level-of-understanding. As can be seen from figure 6.5 there is no correlation between the raw number of classes a subject commented and their completion time of the Measured task (the blue marks represent censored results). When the number of classes documented is used in a uni-variate Cox

Proportional Hazard model the result is $p = 0.916$ which confirms the idea that there is no connection between completion times and number of classes documented. It is only when adding the quality of the comments does any (weak) form of correlation become apparent, but that then becomes highly subjective.

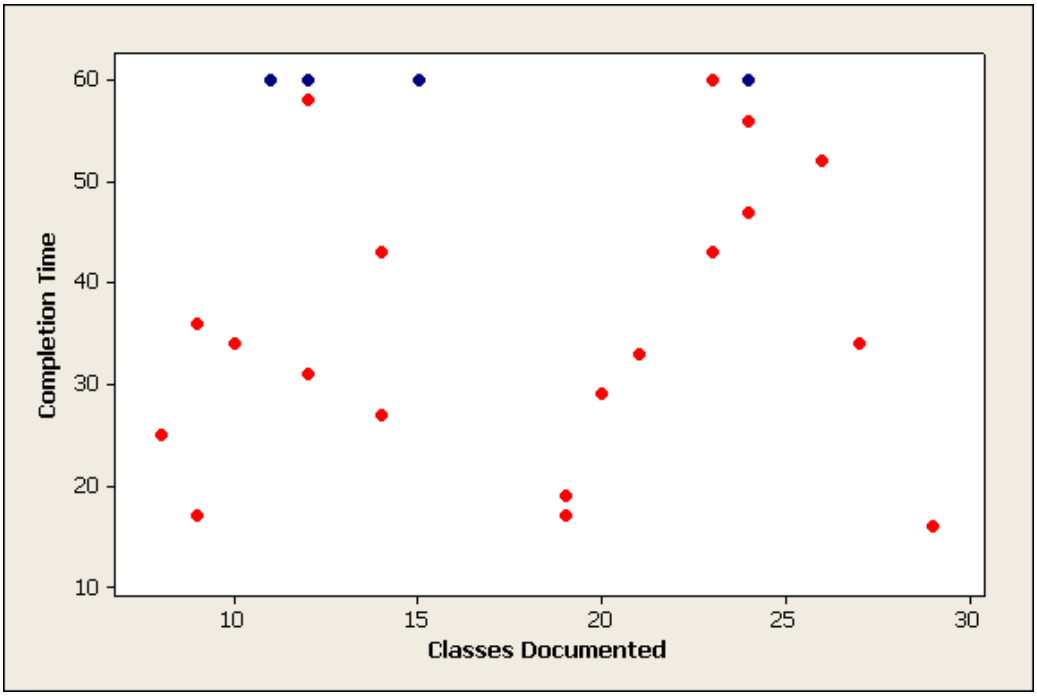

Figure 6.5: Completion Times vs Classes Documented

Completion and Failure

There are a number of non-statistical yet interesting features of the work the the subjects produced. There were 12 subjects who failed to complete the Measured task in the given hour: two of these could be described as being close to finishing, one being down to the stage of fixing typos in his code, while the other had successfully got the new command working for two of the three ranking systems, so clearly had a firm grasp of how the system as a whole worked. However, from examining the code and debriefing the other ten it became clear that those subjects had a *critical* failure in their level-of-understanding of the code. The amount of code that was wrong or needed to be changed to get a working program would be fairly minimal *if only* they had a clear level-of-understanding of how the code worked. Amongst the subjects that succeeded there was a generally consistent manner in which the subjects successfully performed the Measured task. The steps were not necessarily taken in the following order but they consisted of:

1. Locate the correct command generation class and add code by copy and pasting similar code with minor modification

2. Copy and paste an existing single argument command class into a new file, then modify to call correct method in ranking system interface

3. Locate private methods already in ranking systems that perform the work of the Measured task and add a method to the ranking system interface and wrapper code to the ranking systems so that it is accessible by the new command.

Those twn subjects who totally failed to complete in time could be split into two categories: did not understand (and thus could not implement) how a command was generated and called the ranking system; or tried to incorrectly use the built in Java API collections methods to fulfil part 3 rather than using the already existing method. Subjects who failed for the first reason basically tried to do work in the wrong place, placing code that should have been in the ranking systems in the command object or code that should have been in the command object in the command factories and so on. Subjects who failed due to the second reason seemed to get tunnel vision focus on using the Java API and they could not step back and see why it did not work. The subjects who failed to understand how a command was generated could be said to have a very low level-of-understanding of the system: it was not a simple piece of knowledge they were missing but a major chunk of how the system operated. Those who were focused on the Java API issue had a better level-of-understanding and it could be said that they were caught on a technicality, as they understood the bulk of the system. The majority of subjects failed due to not understanding how the command objects acted as communicators between the rankings systems and the command line interface. Both the Enhancement and Document subjects failed in this way with no discernible differences in the incorrect code they produced.

One interesting facet is that a number of subjects located the private methods in the ranking systems (step 3) but did not use them. Some thought that the use of the private methods was somehow inappropriate while others were of the view that as they did not write them they could not be completely sure about how they worked. The majority of these people attempted to write their own version of it, with some of them simply copying and pasting the private methods to make new public versions of them. Unexpectedly, the times of subjects who copy pasted the find method or created their own were not noticeably different from the other subjects, although none of them were in the top 20% of completion times. These subjects were were evenly distributed between the Enhancers and the Documenters.

One interesting case was that of subject 62, who was one of the subjects who were close to finishing. He was able to successfully implement the ISIN command for the League System and Ladder System, but was unable to do so for the Points System. Examining his code revealed that he did not understand how the internals of the Points Ranking system worked, so in an attempt to get the ISIN command to work he took code from the Ladder System (the simplest of the three systems) related to storing names and grafted it onto the Points Ranking system and then tried to use that code to fulfil the ISIN command. Technically it could have worked, but was a vastly complicated and error prone approach to attempt.

Comparing the results of subjects 65 and 66 shows what a difference a level-of-understanding makes. The programmers were comparable, both having an A-grade, and rating them-

selves as a 6 and 7 out of 10 respectively for Java programming ability. From the code produced for the Initial task, it is clear that subject 65 did not fully understand how the CommandFactory inheritance hierarchy worked, while subject 66 was well on the way to completing the UNDO task. This gap in subject 65's level-of-understanding meant that he took 41 minutes to complete the Measured task, while subject 66 took only 16 minutes. However, the code the pair of programmers produced was almost identical, with only trivial differences in identifier names and formatting, both of them having undertaken the same approach to solving the problem once they had obtained the necessary level-of-understanding.

The overall picture drawn from examining the artefacts of the subjects' work is that there were no discernible differences in the work of the subjects based on the Initial task. In many regards this is a surprising result: mental models, such as von Mayrhauser's state that the subjects would have performed different mental actions whilst performing the program understanding necessary for the Initial tasks. However, these differences did not manifest themselves in the Measured task, either in the qualitative examination of the work they produced nor the quantitative examination of the distribution of completion times.

Nature of Subjects

Over the years in which the experiment was run a shift in the subjects' use of the computing environment was noticed. Although the subjects all still had at least one year of Linux experience, Linux itself has changed and this seems to have had an effect on the subjects. The first cohort were almost all Emacs users who used the command line to perform all their actions. The third cohort were much more GUI inclined: they navigated the file system using a file browser rather than the command line and used more GUI friendly text editors. Despite these observed differences the analysis of differences by cohort in section 6.8.3 showed absolutely no significant difference in the completion times nor were there any differences in the type of work produced by the subjects.

6.9 Threats to Validity

6.9.1 Two Enhancements

The greatest threat to validity in the experiment relates to the two Enhancement tasks. If they are too similar then the Enhancement group is being trained to succeed in the Measured task, which distorts its ability to be used as a metric of the level-of-understanding gained. Ideally, the Enhancement tasks should be of different 'types': changeative and additive. The two tasks used are both additive in nature. However, as practising Enhancement is not practising any specific skill other than programming, which is something the subjects are supposed to be able to do, it was not felt that the two Enhancement tasks would significantly bias the results.

6.9.2 Bug

There was a missing feature in the experiment code. Some subjects noticed and indicated the fact that the code to move people in one of the ranking systems in the case of a draw result was missing. Upon its initial discovery, which was after about half the 1st cohort of subjects had done the experiment, it was decided that no action would be taken to remedy it. The code was neither fixed nor was special attention drawn to the bug during the description of the system. The bug was not in an area of code that directly affected the ability of the subjects to complete either the Initial or Measured Enhancement task. Furthermore, as no Documenters got to the level of documenting the method the bug was in, it does not appear that any subject or group of subjects was disadvantaged by the bug. For the further replications of the experiment it was decided not to fix the bug. The justification is that the materials should be totally unaltered to avoid introducing an extra (however small) threat to validity into the experiment.

6.10 Unanswered Questions

The experimental design does not consider what should happen if a subject completes the Initial Enhancement task in less than the given hour. With the materials used this is certainly possible, as demonstrated by a strong post-graduate programmer who was able to complete both the Initial and Measured tasks in under an hour. This is especially pertinent when considering replicating this experiment with industry professionals who can have a wide range of abilities [49, 59].

There are three potential solutions to this problem. The first would be to reformulate the Initial Enhancement task to make it more difficult, thus reducing the chance of anyone being able to finish the task in one hour. The second would be to have a series of Enhancement tasks that can only be performed sequentially, thus providing the best subjects with a stream of Enhancement work. The third and final solution would be to consider it a non-problem and let the subjects read the code for the remaining time. All three approaches have their own benefits and drawbacks drawbacks.

The first option risks confounding the experiment by introducing artificial levels of complexity to the Initial task. Any given task can only get so complex before it stops being a reasonable facsimile of a Perfective maintenance task and starts becoming an esoteric request that requires a detailed level-of-understanding about specific pieces of code to even be understood. As stated in section 5.6 unless specifically testing issues of complexity all materials should be as straight forward as possible. Furthermore, there is no guarantee no matter how difficult the task is that a particularly strong programmer cannot finish it in time unless the task is made unfeasibly large, in which case that would once again introduce the potential to confuse the subjects. The second approach has problems with presenting the work. In the current experimental design, the subjects are given time to read the problem specification and ask questions about it, to avoid confusion. If the complete suite of problems is introduced before the Initial task starts then an amount of time is spent discussing tasks that the majority of subjects will not undertake. The

third approach creates confusion as to what the difference between the two Initial tasks is. While reading of the code must take place to perform the Initial Enhancement task it is always reading for a specific purpose, gaining the information necessary to complete the Enhancement task. Letting the active subjects simply read the code in an undirected manner would introduce to the active task many, many elements of the passive task. This would make it nearly impossible to ascribe any benefits or drawbacks for the active to the fact that they did Enhancement, it could be the case that the undirected reading of the code is what gave them their level-of-understanding. The profound effect this would have on trying to interpret the results of the experiment means that this third option is not a realistic candidate for solving the original problem.

Fundamentally, this is a problem related to having to fit the Initial task into one hour. In a larger experiment, with more time for the Initial task it would be perfectly reasonable to have a number of Enhancement tasks to perform as the amount of time spent reading about the tasks would be small relative to the total amount of time to perform the tasks. This would be an acceptable approach to ensuring the Enhancement subjects were supplied with enough work to last them the full length of the time available for the Initial task.

6.11 Conclusion

This chapter has described, with reference to established literature identified in chapter 5, the design and implementation of a robust experiment created to look at one factor of level-of-understanding issues. The trade-offs and potential confounding factors were identified, described and justified. The result is a design which has strong internal validity. Ways of increasing the external validity are discussed in the following chapter.

The context of the experiment was examining what work a software immigrant should perform when introduced to an environment without mentors. The results did not reveal any discernible difference in the level-of-understanding of subjects performing the active and passive tasks. Furthermore, there was no discernible difference in the quality or approach to the Measured task depending on Initial task, as subjects from both groups made similar bad choices or questionable design decisions. This strongly suggests that software immigrants should start performing active tasks upon joining a maintenance team as they gain just as much of a level-of-understanding as programmers who spend time performing passive work, yet they are producing work of immediate benefit to the system.

Chapter 7

Further Work

7.1 Introduction

There are three main pieces of work in this thesis: the structured literature review, the interviews with maintenance programmers and the experiment. Both the interviews and experiment have the scope for expansion as well as prompting thoughts on alternative avenues for research. This chapter presents thoughts about what work should follow this thesis.

7.2 Interviews

7.2.1 Further Interviews

The replication of previous empirical work is a vital source of information. As Software Engineering is an empirically based field, it is one where changes in the attributes of its practitioners necessitate changes in the focus of research. As a result, regular longitudinal based studies of the elements of Software Maintenance, as advocated by Lientz and Swanson and practiced by Lehman, are required. Of even more interest than identifying those elements that are changing is the thought that the population is not changing. In a field which has undergone large changes in the Products (with the rise of Relational Databases, Object Oriented Programming and now Web Services) and Processes (with iterative and now agile development methodologies) a lack of change in the People and the problems they perceive is a more interesting result than finding out that they have changed. If a lack of change is repeatedly discovered then this suggests that the fruits of Software Maintenance research is either not reaching practitioners or is not being considered useful by them.

Further Question

An important further interview question which can be used to validate the other answers given is: "How is your success as a maintenance programmer measured?". This is an important question as the maintainers' external measure of success must surely influence how they undertake the job. If the principal measure of success is the number of change

requests fulfilled, in an environment where Preventative maintenance does not get the luxury of an official change request document, then this would be expected to affect how much Preventative maintenance the maintainer performs. This is an expectation that should be measured, as if the measure of success does alter the type of work the maintainers undertake, then research that looks at how altering the measure alters the work could well reveal ways of improving the efficiency of groups performing maintenance.

7.2.2 Alternate Survey Methods

Due to solely relying on the answers the interviewees provided, the interviews suffer from a lack of objectivity. The only sure way of producing objective data is through some form of observational, ethnographic study, although they, as previously identified, have their own issues. While setting up ethnographic studies is difficult, due to the inconvenience to the companies and programmers involved as well as the long time scales and arduous nature of quantitatively analysing the produced data, Singer et al. [65] show that there are also strong benefits. An ethnographic study allows researchers to compare what programmers say they do with what they actually do, and allow researchers to make stronger claims about qualitative results.

7.2.3 New Research Directions

Three aspects of the interviews suggest other possible research directions that can be undertaken. The first, relating to the use of program logs, would be complementary to the work presented in the rest of the thesis. The others, involving the difficulty of correctly assigning work to maintenance programmers and the assessment of maintenance 'signatures' would be different fields of research.

Programming Logs

There is scope for investigating important parts of a maintainer's work that were raised by the interviews. In the same way as Taylor et al. [74] looked at maintenance training, other individual aspects of the maintenance process could be considered in greater detail. For example, the greatest point of commonality that was identified between programmers was the use of program logs to aid debugging and understanding of the program. By developing a greater knowledge of exactly what type of information maintenance programmers looked for, strategies could be developed to help them look for that relevant information. The use of logs as a teaching tool could also be examined, as interviewees indicated that they would use them to teach, showing software immigrants the connection between normal behaviour and log output. These and other details of the maintenance process could be very valuable in the compilation of a new maintainer's manual. One possible ultimate aim of this research could be the development of some form of expert system which would help diagnose the possible cause of bugs without recourse to an expert maintenance programmer.

Classification of Maintenance Signatures

There is a strong suggestion, both from my interviews and the literature, that various artefacts of a company's maintenance process are heavily dependent on the type of Product being maintained and the structure of the team maintaining it. For instance, it is noticeable that those programmers who maintained programs internal to the company performed fewer bug fixes and more perfective maintenance than programmers who maintained highly configurable Products or Products that were sold or used by external consumers. As a result, general maintenance advice, based on the average figures produced from general surveys, will be of variable levels of utility to companies and programmers depending on how closely their Product and teams match the "average". Indeed the "average" might well be a completely misleading idea to which no actual maintenance group conforms. For example, the LS survey had standard deviations of around 22 points on a 100 point scale for some answers, which is a very large level of variability. As a result, it may well be worthwhile to try to discover factors in organisational issues, program design and domain areas that affect the maintenance 'signature' of systems: the *pattern* of work performed on a system. With more formal classifications, it may be possible to determine positive and negative aspects of their composition that allow a more accurate comparison between different companies' systems. So, for example, one group may be maintaining a *Small*, *High Configurability*, *Young*, *Object Oriented*, *Internally Deployed* system whilst another another has a *Very Large, Linear, Old, Procedural, Commercially Deployed* system. The differences in the attributes of the maintenance of the two systems (such as the proportion of Adaptive, Corrective, Perfective, Preventative maintenance) could potentially be predicted based on the differing properties of the systems.

Work Assignment

One of the major points identified from the interviews is that: *"To a varying degree, the accurate assignment of responsibility for a bug fix or feature upgrade is a problem"*. Any alternate process could only be properly tested by performing change to a company's maintenance process. On identifying a company that does not have a formal bug assignment process, and whose programmers identify this as a problem, the following steps could be taken:

- Observe the bug fixing efficiency of the team, taking into account bug types, team size and overt process issues.
- Design and implement a more formal bug assignment process that meshes with their current maintenance process.
- After the new assignment process has been used for a long enough period, once again measure the team's bug fixing efficiency.

Obviously, persuading even one team within a company to change its process is a difficult task, as there could well be several managerial staff who would be accountable if the experiment failed, but who would not be rewarded if it succeeded. However, as a software engineering researcher I cannot be satisfied with hypothesis alone, and only through the practical application of ideas can I find out if mooted improvements are of any use.

7.3 Experiment

7.3.1 External Replication

It is important that the experiment and results are replicated by an external research group. As Carver et al. [14] state, there are many implicit assumptions that go into experiment design that are revealed when other groups, with their own assumptions, try and replicate an experiment. As such, any external replication may reveal some aspect of my own execution of the experiment that may have inhibited or boosted either of the two treatment groups. Successful replication of the experiment, with the same results, would greatly increase the internal validity of the experiment.

7.3.2 Attempt to Replicate Results With Different Materials

The exact composition of the materials of the experiment are a confounding factor. It could be that the precise Initial or Measured task, or indeed the program itself, affected the results in some way, either inhibiting or boosting one or other of the groups. To improve the generalisability of the results, the experiment should be repeated with alternate materials. The single most important change would be to make the Measured task changeative in nature to help reduce the possibility that the similarity of Enhancement tasks gives benefit to the Enhancement group.

7.3.3 Attempt to Replicate Results With Different Types of Subjects

Another method to increase the generalisability of the experiment would be the use of experienced industry practitioners as subjects rather than undergraduates. There is the possibility that industry practitioners give different results than students. That the professionals' greater innate levels of programming ability would result in the the different tasks affecting the subjects in different ways and producing statistically different completion times for the Measured task. If this were the case then it would be interesting to examine the tasks from a program understanding perspective to try an analyse the different cognitive actions they make the subjects perform. Furthermore, if the use of professional programmers results in the Documenting task being faster than the Enhancement task then this would revise the recommendation drawn from the experiment.

The increased ability level of industry practitioners would necessitate a change in the materials. As section 6.10 highlighted, the completion of both the Enhancement tasks is well within the capabilities of a strong programmer. To use industry practitioners would mean having to increase the size, and possibly difficulty, of the code and tasks, although without significantly increasing the domain complexity of either. The choice of domain would also be of greater importance, as more experienced programmers would be likely to have more in-depth knowledge of various domains. Selecting a domain that, say, half of the subjects were intimately familiar with would, of course, risk confounding the experiment. However, given that an experiment using professionals would be longer, due to the fact that they would have to be paid at least a day's wage, there would be more

time for the teaching of the chosen experiment domain to try to eliminate any differences that may exist.

7.3.4 Mentoring and the Base Assumptions

The experiment was constructed to answer the question of what software immigrants should do when faced with a situation without mentors, the base assumption being that mentoring is the best available method outside of unimplemented formal training methodologies for gaining a level-of-understanding about a system. This experiment design can be simply adapted to examine whether mentoring is a better approach. Instead of Enhancing or Documenting the code as the Initial task, the subjects could be Mentored instead. Mentoring, as a concept, covers a large number of different approaches. At one extreme mentoring covers programmers who reside as oracle type figures, taking no action until asked by the software immigrants, at which point they will dispense advice. At the other extreme there are mentors who fully guide the software immigrants, taking them on a guided tour of the system, pointing out the interesting places as they go, and being the main driving force behind what happens during the mentoring. There is also a wide spectrum inbetween. Any experiment involving mentoring would have to carefully define the mentors role to allow any form of replication.

7.3.5 Other Task Types

In effect, the experiment design acts as a framework for examining level-of-understanding issues. The Initial tasks can be anything that the experimenter wants, so the subjects could be using two different tools to perform the same enhancement, different code inspection techniques, or refactoring approaches: the list is practically limitless. By consistently using this framework with the same code and Measured task, a researcher's knowledge of comparative level-of-understanding issues could be reliably built up over a period of years. This is another benefit of the experiment being of a between-groups nature: each additional Initial task that is added only increases the number of treatment groups by one. In a within or pseudo-within groups design, every additional Initial task would require the creation of multiple treatment groups to compare the relative effects of it against other approaches.

Furthermore, replicating this experiment with professional programmers and the larger time scale that would imply, may make it feasible to use the other maintenance types (Adaptive, Corrective, Preventative) as an Initial task. Part of the reason of rejecting some of the tasks for the implemented experiment was due to the limited amount of time the subjects would have to work with the code. In a lengthier experiment it would become more practical to introduce these alternate task types.

7.4 Further Analysis of Software Immigrants Work

From the interviews and the analysis of the literature on software immigrants, it seems that software immigrants undertake a variety of different approaches to building a level-of-understanding about a sub-system. The results from the experiment suggest that, in an

environment that lacks external sources of information, software immigrants should start working with the code as soon as possible. The limited results from Singer et al. suggest that it takes some time before software immigrants start working with the code and making changes, so more investigation is needed as to exactly what activities software immigrants undertake and how they affect the software immigrants' level-of-understanding. Particular attention should be paid to the effects that the different maintenance tasks have on a software immigrant's level-of-understanding. Only one of the four types of maintenance (Adaptive, Corrective, Perfective and Preventative) was examined by the experiment. It might be that once some knowledge of the system has been built up, different tasks have different effects on further gains of level-of-understanding. As a result, it is important to look at all the activities the software immigrants perform while trying to gain a level-of-understanding about the sub-system.

The goal of the research would possibly be to suggest ways for software immigrants to self-organise the maintenance work they have to perform to maximise the level-of-understanding gains about a sub-system. The reason the research would be examining ways of self-organising the work would be that, if it was organised by another programmer or manager, then that would suggest that such an organiser is available and they would therefore fulfil the mentoring role. This is not to suggest that there is no benefit in a mentor organising the work of software immigrants; infact, part of the results of Berlin [6] were that the mentors beneficially controlled the early work of the software immigrants in the company. However, given the assumption that no mentor is available it would not be relevant whether or not their organisation of a software immigrant's workload would be beneficial.

It may well be that the recommendations as to how software immigrants should organise their work load would be generic in nature, applicable to software immigrants across a variety of companies and software Product types. This would be a useful light-weight approach to aiding software immigrants. Light-weight approaches, ones that do not rely on the production of many materials that need to be kept current with the system, are particularly useful, as research shows low-turnover environments with few software immigrants are the norm. This means that heavy-weight solutions do not get implemented, as they require too much time investment when software immigrants are not a frequent occurrence. A light-weight, generic approach could be taught in undergraduate software engineering courses or could be introduced into companies' basic introductory courses.

Chapter 8

Conclusion

8.1 Summary of Work

The thesis is founded on a systematic review of the literature. General reading had suggested that there was a lack of both empirical papers and specifically papers about People in Software Maintenance. To investigate this possibility a structured, research trends based, review of the literature was performed. This involved reading, to varying degrees of detail, 871 papers from three publications and classifying them into four categories. This review revealed a similar level of empirical work in Software Maintenance as there is in Software Engineering as a whole. As the volume of empirical papers in Software Engineering had been deemed insufficient, this meant that empirical work in Software Maintenance is also insufficient. The survey also revealed that the proportion of papers dealing with People in Software Maintenance was far below the expected level. This was determined to be a gap in the research in the field.

This naturally lead to performing work to try and fill this gap in the research. Of the available options, interviews were considered the most appropriate manner in which to gather information. As a result a set of interview questions were formulated and maintenance programmers from multiple companies were questioned. The interview questions and their style were based on two papers by Singer (et al.). Although the interviews were based on two pieces of previous work they are not simply a replication of that work. The Singer et al. study focused on very specific needs, examining the work practices of a single set of programmers in a single team within a single company. The Singer study was a much more broad based view, covering basic impressions at a number of companies. My own study was at a level of detail sitting somewhere between the two: whilst it had a broad (yet shallow) base like the Singer study, it provided opportunities to go into great detail on topics related to information gathering strategies. Basing the questions on the two previous surveys did allow some form of valuable replication and essential comparison of results. By partly replicating previous work in the field and discovering, in the areas replicated, broadly similar qualitative and quantitative results, a firm basis for future work was established. This similarity of results in the replicated work provides support for suggesting the generalisability of the non-replicated work.

The interviews, and the Singer study, suggested that there was a problem for software immigrants in maintenance. A lack of sources of information about the code and no institutional policies towards teaching the immigrants would result in a very challenging environment. To ascertain whether this was the case, a further literature review was undertaken, this time focusing on papers that examined issues relevant to software immigrants.

There were two major results from this literature review. The first result was that problems of staff turnover were considerably less common than "common knowledge" about software maintenance suggests. Examining group studies of different software maintenance groups showed that very few of them rated turnover as a significant problem. Conversely, studies of individual companies suffering high turnover, which rate high turnover as a problem, show that when it is an issue it is a *major* issue, possibly even the overriding problem that the maintenance team suffers from. As the majority of companies are low-turnover environments, this feeds into poor documentation. As maintainers specialise in specific sub-systems, any documentation they produce is primarily read by no-one other than themselves until a software immigrant has to take responsibility for that part of the system. The programmers see no need to document their knowledge in this manner and this results in poor documentation.

The second result was concordant with the results of my interviews: very few companies have training methodologies in place to aid software immigrants develop a level-of-understanding about the system they are working on. What informal training does take place relies almost exclusively on the existence of sub-system experts, mentors, to be used as an information source. However, as my interviews had shown, such mentors are not always available: a result which would firmly strand software immigrants on their own when it comes to developing a level-of-understanding about their area of responsibility.

Given the expense of developing formal training methodologies, and the low level of use they would receive due to the general low level of turnover that most maintenance teams have, it was thought worthwhile to examine alternate, mentor-free, methods for software immigrants to gain a level-of-understanding about a system. The decision was made to examine *work based* approaches to developing a level-of-understanding about a system. The method of analysis used was a controlled laboratory experiment. This method was chosen to provide quantitative data for the thesis, to sit along side the qualitative work of the interviews. The two work approaches considered were documenting and enhancing a program. Alternative tasks were considered and rejected for reasons of both practicality and belief that they would not sufficiently exercise a subject's level-of-understanding about the code. From the wide variety of approaches available to measure the subjects' level-of-understanding about the code, an indirect method, using time to completion of a programming task was chosen. Other approaches were rejected, as, while they were a more direct means of measuring a subject's level-of-understanding, they either introduced subjective analysis, which the experiment design had otherwise strived to avoid so that it could retain its purely quantitative nature, or they introduced large confounding factors that could not be controlled.

To produce reliable results the experiment was run on three separate occasions to get a total of 53 subjects. Survival Analysis, a robust statistical technique that has seen little use in Software Engineering literature, was used to analyse the results. Somewhat surprisingly, there was found to be no significant statistical difference between the two treatment groups. This meant that there was no additional benefit to undertaking the passive rather than active task. Further qualitative analysis, based on interviews with the subjects and an analysis of the work they produced for both the Initial and Measured task, also showed no noticeable difference in the approaches used on the manner of failure observed. Given this result, it means that in the hostile environment that software immigrants often find themselves they may as well start fulfilling maintenance requests as soon as possible as they gain no additional benefit from taking a more hands off high level view. This view also holds true from the company's perspective: in a hostile environment it is better for the company to get the software immigrants to start work as soon as possible.

Finally, I have identified how the major components of the work can be improved and extended. The interviews can be increased in volume and combined with other investigative techniques to cross check the validity of their data. The experiment can be altered in the size and complexity of the materials involved, and could also use professional programmers as the subjects to increase the generalisability of the results.

8.2 Measures of Success

The four measures of success were defined in the introduction are as follows:

- Present evidence on the current state of mainstream software maintenance research in relation to empirical research to show a gap in the literature
- Perform empirical work to help close the gap
- Identify unresearched problems of software immigrants
- Compare and contrast different approaches for software immigrants to develop a level-of-understanding about a system

Chapter 2 presented the results of a systematic review of mainstream software maintenance research. Mainstream research was defined as the premier journal and conference in the field plus a slightly smaller regional conference. Similar surveys had been done in the fields of Computing Science and Software Engineering, both of which concluded that there was insufficient empirical work in the field. The results of my survey were broadly in line with these previous surveys. Furthermore, given the basic taxonomy of Software Maintenance into People, Processes and Products, the survey showed that only 8% of papers contained notable work on People in Software Maintenance. This was considered far below what should be expected.

In order to help to close this gap, a series of interviews with maintenance programmers was undertaken, presented in chapter 3. This work, being both empirical in nature and

dealing with People in Software Maintenance, neatly addressed the gap in research identified. The interviews consisted of two primary components: the first tried to get a generalised overview of how maintenance is performed trying to identify good and bad features of maintenance as it happens, while the second, more in-depth section examined how maintainers gathered and used information to perform maintenance. A large amount of interesting data was uncovered but the overriding impression gained was of a lack of institutionalised training for software immigrants as well as a lack of any sources of information about the system being maintained except for the system itself. This presented a very challenging environment for software immigrants and an avenue for further, more detailed research.

Finding the topic of software immigrants the most potentially fruitful line of enquiry, I undertook further research to try and determine the characteristics and problems of software immigrants. A further literature review was performed which complemented the findings of the interviews, presenting a picture of the typical company as one that does not have any defined training process for aiding software immigrants gain a level-of-understanding about the system they have to maintain. The literature also does not suggest anything in the way of validated solutions to this problem. An assumption of the existence of mentors was discovered but the interviews demonstrated that mentors were not always available. There were suggestions for formal training methodologies, however even these implicitly involved the use of mentors whilst at the same time requiring a large time investment in the production of materials. The Pigoski and Looney [54] approach (in an environment without mentor but with some reasonable quality documentation) used standard maintenance tasks as one of the principal ways of driving maintainers to increase their level-of-understanding. However no *comparison* of the utility of the tasks was presented.

As the material on software immigrants was lacking, a comparative analysis was required to measure the effect that different tasks have on a subject's level-of-understanding. To meet this goal an experiment was carefully designed, utilising the experimental literature and piloting of potential designs, tasks and ideas. The experiment was run three times to gain a sufficient number of subjects for robust statistical analysis. The experiment examined two work based, non-mentor, approaches to gaining a level-of-understanding about a piece of code. One task involved actively working with the code whilst the other involved the subject being passively hands-off. The final results showed no difference in the subject's level-of-understanding by undertaking active or passive tasks. Further qualitative analysis also showed no noticeable difference between the two groups. Given that there is no benefit to the passive work, then in the challenging environment previously identified, without mentor or external sources of information, this experiment suggests that software immigrants should be placed straight to work upon joining a team, as they gain no benefit from not going straight to work.

8.3 Answering the Thesis Statement

The state-of-practice can be a hostile place for the software immigrant performing maintenance. By means of an extensive literature review and empirical study of maintainers I have demonstrated that the environment is often one without trustworthy documentation or experienced sub-system experts to consult. By means of a controlled laboratory experiment I have shown that in such an environment the software immigrant gains no benefit from taking a passive approach to the code and as such should start working with the code, fulfilling maintenance requests, as soon as possible.

The specific hypothesis tested to determine if there was any difference for software immigrants between undertaking passive and active tasks with the code is as follows:

Null Hypothesis 1: There is no difference in the level-of-understanding of the Enhancement and Document groups.

Testing for differences between the two groups gave a result of $p = 0.717$ which means there is no statistically significant difference between the groups. As Enhancement represented an active task and Documenting represented a passive task there is no significant difference between and active and passive approach to working with a system.

Appendix A

Experimental Statistics

A.1 Kaplan-Meier Survival Curves

A.1.1 By Group

```
Variable: Time
Group = D

Censoring Information  Count
Uncensored value          20
Right censored value       5

Censoring value: Censor = 1


Nonparametric Estimates


Characteristics of Variable

            Standard   95.0% Normal CI
Mean(MTTF)     Error    Lower    Upper
     41.16   3.20772  34.8730  47.4470

Median = 43
IQR = 29  Q1 = 29  Q3 = 58


Kaplan-Meier Estimates

       Number  Number     Survival   Standard   95.0% Normal CI
Time  at Risk  Failed  Probability      Error     Lower    Upper
  16       25       1         0.96  0.0391918  0.883185  1.00000
```

17	24	2	0.88	0.0649923	0.752617	1.00000
19	22	1	0.84	0.0733212	0.696293	0.98371
25	21	1	0.80	0.0800000	0.643203	0.95680
27	20	1	0.76	0.0854166	0.592586	0.92741
29	19	1	0.72	0.0897998	0.543996	0.89600
31	18	1	0.68	0.0932952	0.497145	0.86286
33	17	1	0.64	0.0960000	0.451843	0.82816
34	16	2	0.56	0.0992774	0.365420	0.75458
36	14	1	0.52	0.0999200	0.324160	0.71584
43	13	2	0.44	0.0992774	0.245420	0.63458
47	11	1	0.40	0.0979796	0.207964	0.59204
52	10	2	0.32	0.0932952	0.137145	0.50286
56	8	1	0.28	0.0897998	0.103996	0.45600
58	7	1	0.24	0.0854166	0.072586	0.40741
60	6	1	0.20	0.0800000	0.043203	0.35680

Distribution Analysis: Time by Group

Variable: Time
Group = E

Censoring Information	Count
Uncensored value	21
Right censored value	7

Censoring value: Censor = 1

Nonparametric Estimates

Characteristics of Variable

	Standard	95.0% Normal CI	
Mean(MTTF)	Error	Lower	Upper
36.2857	3.48781	29.4497	43.1217

Median = 30
IQR = 38 Q1 = 21 Q3 = 59

Kaplan-Meier Estimates

Number	Number	Survival	Standard	95.0% Normal CI

Time	at Risk	Failed	Probability	Error	Lower	Upper
9	28	1	0.964286	0.0350707	0.895548	1.00000
10	27	1	0.928571	0.0486704	0.833179	1.00000
12	26	1	0.892857	0.0584512	0.778295	1.00000
16	25	1	0.857143	0.0661300	0.727530	0.98676
19	24	2	0.785714	0.0775443	0.633730	0.93770
21	22	1	0.750000	0.0818317	0.589613	0.91039
23	21	1	0.714286	0.0853735	0.546957	0.88161
24	20	3	0.607143	0.0922962	0.426246	0.78804
26	17	1	0.571429	0.0935220	0.388129	0.75473
28	16	1	0.535714	0.0942498	0.350988	0.72044
30	15	2	0.464286	0.0942498	0.279560	0.64901
39	13	2	0.392857	0.0922962	0.211960	0.57375
41	11	1	0.357143	0.0905522	0.179664	0.53462
51	10	1	0.321429	0.0882594	0.148443	0.49441
52	9	1	0.285714	0.0853735	0.118385	0.45304
59	8	1	0.250000	0.0818317	0.089613	0.41039

Distribution Analysis: Time by Group

Comparison of Survival Curves

Test Statistics

Method	Chi-Square	DF	P-Value
Log-Rank	0.131778	1	0.717
Wilcoxon	0.930625	1	0.335

A.1.2 By Grade

Variable: Time
Grade = a

Censoring Information	Count
Uncensored value	25
Right censored value	4

Censoring value: Censor = 1

Nonparametric Estimates

Characteristics of Variable

Mean(MTTF)	Standard Error	95.0% Normal CI Lower	Upper
31.1379	3.00403	25.2501	37.0257

Median = 27
IQR = 22 Q1 = 19 Q3 = 41

Kaplan-Meier Estimates

Time	Number at Risk	Number Failed	Survival Probability	Standard Error	95.0% Normal CI Lower	Upper
9	29	1	0.965517	0.0338830	0.899108	1.00000
10	28	1	0.931034	0.0470544	0.838810	1.00000
12	27	1	0.896552	0.0565523	0.785711	1.00000
16	26	2	0.827586	0.0701445	0.690106	0.96507
17	24	2	0.758621	0.0794627	0.602877	0.91436
19	22	3	0.655172	0.0882632	0.482180	0.82817
23	19	1	0.620690	0.0901022	0.444093	0.79729
24	18	1	0.586207	0.0914572	0.406954	0.76546
25	17	1	0.551724	0.0923495	0.370722	0.73273
26	16	1	0.517241	0.0927925	0.335372	0.69911
27	15	1	0.482759	0.0927925	0.300889	0.66463
28	14	1	0.448276	0.0923495	0.267274	0.62928
29	13	1	0.413793	0.0914572	0.234540	0.59305
33	12	1	0.379310	0.0901022	0.202713	0.55591
34	11	2	0.310345	0.0859091	0.141966	0.47872
39	9	1	0.275862	0.0829961	0.113193	0.43853
41	8	1	0.241379	0.0794627	0.085635	0.39712
43	7	1	0.206897	0.0752216	0.059465	0.35433
51	6	1	0.172414	0.0701445	0.034933	0.30989
52	5	1	0.137931	0.0640329	0.012429	0.26343

Distribution Analysis: Time by Grade

Variable: Time
Grade = b

Censoring Information	Count
Uncensored value	10

Right censored value 2

Censoring value: Censor = 1

Nonparametric Estimates

Characteristics of Variable

Mean(MTTF)	Standard Error	95.0% Normal CI Lower	Upper
43.8333	4.04285	35.9095	51.7572

Median = 43
IQR = 26 Q1 = 30 Q3 = 56

Kaplan-Meier Estimates

Time	Number at Risk	Number Failed	Survival Probability	Standard Error	95.0% Normal CI Lower	Upper
21	12	1	0.916667	0.079786	0.760290	1.00000
24	11	1	0.833333	0.107583	0.622475	1.00000
30	10	1	0.750000	0.125000	0.505005	0.99500
36	9	1	0.666667	0.136083	0.399949	0.93338
39	8	1	0.583333	0.142319	0.304394	0.86227
43	7	1	0.500000	0.144338	0.217104	0.78290
47	6	1	0.416667	0.142319	0.137727	0.69561
52	5	1	0.333333	0.136083	0.066616	0.60005
56	4	1	0.250000	0.125000	0.005005	0.49500
58	3	1	0.166667	0.107583	0.000000	0.37753

Distribution Analysis: Time by Grade

Variable: Time
Grade = c

Censoring Information	Count
Uncensored value	5
Right censored value	6

Censoring value: Censor = 1

Nonparametric Estimates

Characteristics of Variable

Mean(MTTF)	Standard Error	95.0% Normal CI Lower	Upper
51.2727	4.76751	41.9286	60.6169

Median = *
IQR = * Q1 = 31 Q3 = *

Kaplan-Meier Estimates

Time	Number at Risk	Number Failed	Survival Probability	Standard Error	95.0% Normal CI Lower	Upper
24	11	1	0.909091	0.086678	0.739204	1.00000
30	10	1	0.818182	0.116291	0.590255	1.00000
31	9	1	0.727273	0.134282	0.464086	0.99046
59	8	1	0.636364	0.145041	0.352089	0.92064
60	7	1	0.545455	0.150131	0.251202	0.83971

Distribution Analysis: Time by Grade

Variable: Time
Grade = e

Censoring Information	Count
Uncensored value	1

Censoring value: Censor = 1

Nonparametric Estimates

Characteristics of Variable

Mean(MTTF)	Standard Error	95.0% Normal CI Lower	Upper
52	*	*	*

```
Median = 52
IQR = 0  Q1 = 52  Q3 = 52


Kaplan-Meier Estimates

                                                  95.0% Normal
       Number  Number     Survival  Standard       CI
Time  at Risk  Failed  Probability     Error  Lower  Upper
  52        1       1            0         0      0      0


Distribution Analysis: Time by Grade

Comparison of Survival Curves


Test Statistics

Method    Chi-Square  DF  P-Value
Log-Rank     10.4584   3    0.015
Wilcoxon     12.4451   3    0.006
```

A.1.3 By Cohort

```
Variable: Time
Year = 1

Censoring Information  Count
Uncensored value          15
Right censored value       5

Censoring value: Censor = 1


Nonparametric Estimates


Characteristics of Variable

            Standard   95.0% Normal CI
Mean(MTTF)     Error    Lower    Upper
      37.4   4.05916  29.4442  45.3558
```

Median = 31
IQR = 34 Q1 = 24 Q3 = 58

Kaplan-Meier Estimates

Time	Number at Risk	Number Failed	Survival Probability	Standard Error	95.0% Normal CI Lower	Upper
9	20	1	0.95	0.048734	0.854483	1.00000
12	19	1	0.90	0.067082	0.768522	1.00000
16	18	1	0.85	0.079844	0.693509	1.00000
19	17	1	0.80	0.089443	0.624695	0.97530
24	16	2	0.70	0.102470	0.499163	0.90084
27	14	1	0.65	0.106654	0.440963	0.85904
29	13	1	0.60	0.109545	0.385297	0.81470
30	12	1	0.55	0.111243	0.331968	0.76803
31	11	1	0.50	0.111803	0.280869	0.71913
34	10	1	0.45	0.111243	0.231968	0.66803
36	9	1	0.40	0.109545	0.185297	0.61470
47	8	1	0.35	0.106654	0.140963	0.55904
52	7	1	0.30	0.102470	0.099163	0.50084
58	6	1	0.25	0.096825	0.060227	0.43977

Distribution Analysis: Time by Year

Variable: Time
Year = 2

Censoring Information	Count
Uncensored value	14
Right censored value	4

Censoring value: Censor = 1

Nonparametric Estimates

Characteristics of Variable

Mean(MTTF)	Standard Error	95.0% Normal CI Lower	Upper
38.7222	4.14104	30.6059	46.8385

Median = 34
IQR = 38 Q1 = 21 Q3 = 59

Kaplan-Meier Estimates

Time	Number at Risk	Number Failed	Survival Probability	Standard Error	95.0% Normal CI Lower	Upper
17	18	2	0.888889	0.074074	0.743706	1.00000
19	16	2	0.777778	0.097991	0.585719	0.96984
21	14	1	0.722222	0.105572	0.515305	0.92914
24	13	1	0.666667	0.111111	0.448893	0.88444
25	12	1	0.611111	0.114904	0.385903	0.83632
33	11	1	0.555556	0.117121	0.326002	0.78511
34	10	1	0.500000	0.117851	0.269016	0.73098
39	9	1	0.444444	0.117121	0.214891	0.67400
43	8	1	0.388889	0.114904	0.163680	0.61410
51	7	1	0.333333	0.111111	0.115560	0.55111
56	6	1	0.277778	0.105572	0.070861	0.48469
59	5	1	0.222222	0.097991	0.030164	0.41428

Distribution Analysis: Time by Year

Variable: Time
Year = 3

Censoring Information	Count
Uncensored value	12
Right censored value	3

Censoring value: Censor = 1

Nonparametric Estimates

Characteristics of Variable

Mean(MTTF)	Standard Error	95.0% Normal CI Lower	Upper
40	4.46074	31.2571	48.7429

Median = 41

IQR = 34 Q1 = 26 Q3 = 60

Kaplan-Meier Estimates

Time	Number at Risk	Number Failed	Survival Probability	Standard Error	95.0% Normal CI Lower	Upper
10	15	1	0.933333	0.064406	0.807100	1.00000
16	14	1	0.866667	0.087771	0.694639	1.00000
23	13	1	0.800000	0.103280	0.597576	1.00000
26	12	1	0.733333	0.114180	0.509545	0.95712
28	11	1	0.666667	0.121716	0.428107	0.90523
30	10	1	0.600000	0.126491	0.352082	0.84792
39	9	1	0.533333	0.128812	0.280866	0.78580
41	8	1	0.466667	0.128812	0.214199	0.71913
43	7	1	0.400000	0.126491	0.152082	0.64792
52	6	2	0.266667	0.114180	0.042878	0.49046
60	4	1	0.200000	0.103280	0.000000	0.40242

Distribution Analysis: Time by Year

Comparison of Survival Curves

Test Statistics

Method	Chi-Square	DF	P-Value
Log-Rank	0.008772	2	0.996
Wilcoxon	0.117772	2	0.943

A.1.4 By Self-Rating

Variable: Time
Self-Rating <= 6

Censoring Information	Count
Uncensored value	13
Right censored value	8

Censoring value: Censor = 1

Nonparametric Estimates

Characteristics of Variable

Mean(MTTF)	Standard Error	95.0% Normal CI Lower	Upper
44.0952	3.06726	38.0835	50.1070

Median = 52
IQR = * Q1 = 31 Q3 = *

Kaplan-Meier Estimates

Time	Number at Risk	Number Failed	Survival Probability	Standard Error	95.0% Normal CI Lower	Upper
17	21	1	0.952381	0.046471	0.861299	1.00000
23	20	1	0.904762	0.064056	0.779214	1.00000
24	19	1	0.857143	0.076360	0.707479	1.00000
26	18	1	0.809524	0.085689	0.641576	0.97747
30	17	1	0.761905	0.092943	0.579740	0.94407
31	16	1	0.714286	0.098581	0.521071	0.90750
36	15	1	0.666667	0.102869	0.465047	0.86829
39	14	1	0.619048	0.105971	0.411348	0.82675
41	13	1	0.571429	0.107990	0.359772	0.78308
51	12	1	0.523810	0.108985	0.310203	0.73742
52	11	2	0.428571	0.107990	0.216915	0.64023
56	9	1	0.380952	0.105971	0.173253	0.58865

Distribution Analysis: Time by Self-Rating

Variable: Time
Self-Rating >= 7

Censoring Information	Count
Uncensored value	28
Right censored value	4

Censoring value: Censor = 1

Nonparametric Estimates

Characteristics of Variable

Mean(MTTF)	Standard Error	95.0% Normal CI Lower	Upper
33.9688	3.04159	28.0073	39.9302

Median = 29
IQR = 28 Q1 = 19 Q3 = 47

Kaplan-Meier Estimates

Time	Number at Risk	Number Failed	Survival Probability	Standard Error	95.0% Normal CI Lower	Upper
9	32	1	0.96875	0.0307578	0.908466	1.00000
10	31	1	0.93750	0.0427908	0.853632	1.00000
12	30	1	0.90625	0.0515270	0.805259	1.00000
16	29	2	0.84375	0.0641862	0.717947	0.96955
17	27	1	0.81250	0.0689981	0.677266	0.94773
19	26	3	0.71875	0.0794804	0.562971	0.87453
21	23	1	0.68750	0.0819382	0.526904	0.84810
24	22	2	0.62500	0.0855816	0.457263	0.79274
25	20	1	0.59375	0.0868207	0.423584	0.76392
27	19	1	0.56250	0.0876951	0.390621	0.73438
28	18	1	0.53125	0.0882155	0.358351	0.70415
29	17	1	0.50000	0.0883883	0.326762	0.67324
30	16	1	0.46875	0.0882155	0.295851	0.64165
33	15	1	0.43750	0.0876951	0.265621	0.60938
34	14	2	0.37500	0.0855816	0.207263	0.54274
39	12	1	0.34375	0.0839617	0.179188	0.50831
43	11	2	0.28125	0.0794804	0.125471	0.43703
47	9	1	0.25000	0.0765466	0.099972	0.40003
52	8	1	0.21875	0.0730792	0.075517	0.36198
58	7	1	0.18750	0.0689981	0.052266	0.32273
59	6	1	0.15625	0.0641862	0.030447	0.28205
60	5	1	0.12500	0.0584634	0.010414	0.23959

Distribution Analysis: Time by Self-Rating

Comparison of Survival Curves

Test Statistics

Method	Chi-Square	DF	P-Value
Log-Rank	5.65624	1	0.017
Wilcoxon	6.01127	1	0.014

A.2 Cox Proportional Hazard Model

A.2.1 Uni-Variate Self-Rating

Variables in the Equation

		B	SE	Wald	df	Sig.	Exp(B)
Step 1	SelfRating	.300	.130	5.325	1	.021	1.349

A.2.2 Multi-Variate

Variables in the Equation

		B	SE	Wald	df	Sig.	Exp(B)
Step 1	Grade			9.102	3	.028	
	Grade(1)	.613	1.025	.358	1	.550	1.846
	Grade(2)	.065	1.051	.004	1	.951	1.067

Variables not in the Equation(a)

		Score	df	Sig.
Step 1	SelfRating	1.996	1	.158
	Cohort	.164	2	.921
	Cohort(1)	.037	1	.847
	Cohort(2)	.163	1	.686
	Group	.538	1	.463

A.3 Number of Classes Commented

	Score	df	Sig.
Classes Documetned	.011	1	.916

Appendix B

Demonstration Example

B.1 Survival Analysis

```
Variable: Time
Group = Procedural

Censoring Information  Count
Uncensored value          13
Right censored value       2

Censoring value: Censor = 1

Nonparametric Estimates

Characteristics of Variable

            Standard   95.0% Normal CI
Mean(MTTF)     Error    Lower    Upper
   40.1431   3.07425  34.1176  46.1685

Median = 36.3461
IQR = 20.4016  Q1 = 31.1636  Q3 = 51.5652


Kaplan-Meier Estimates

          Number  Number     Survival  Standard   95.0% Normal CI
   Time  at Risk  Failed  Probability     Error     Lower    Upper
23.0000       15       1     0.933333  0.064406  0.807100  1.00000
27.2854       14       1     0.866667  0.087771  0.694639  1.00000
30.6243       13       1     0.800000  0.103280  0.597576  1.00000
31.1636       12       1     0.733333  0.114180  0.509545  0.95712
32.5210       11       1     0.666667  0.121716  0.428107  0.90523
```

```
34.2221       10       1     0.600000  0.126491  0.352082  0.84792
34.5714        9       1     0.533333  0.128812  0.280866  0.78580
36.3461        8       1     0.466667  0.128812  0.214199  0.71913
39.0325        7       1     0.400000  0.126491  0.152082  0.64792
40.3510        6       1     0.333333  0.121716  0.094774  0.57189
45.2102        5       1     0.266667  0.114180  0.042878  0.49046
51.5652        4       1     0.200000  0.103280  0.000000  0.40242
56.2528        3       1     0.133333  0.087771  0.000000  0.30536

Distribution Analysis: NTime by NGroup

Variable: Time
Group = ObjectOriented

Censoring Information  Count
Uncensored value          10
Right censored value       5

Censoring value: Censor = 1

Nonparametric Estimates

Characteristics of Variable

            Standard   95.0% Normal CI
Mean(MTTF)     Error    Lower    Upper
   37.1333   5.11890  27.1005  47.1662

Median = 37
IQR = *  Q1 = 15  Q3 = *

Kaplan-Meier Estimates

       Number  Number     Survival  Standard   95.0% Normal CI
Time  at Risk  Failed  Probability     Error     Lower    Upper
  11       15       1     0.933333  0.064406  0.807100  1.00000
  12       14       2     0.800000  0.103280  0.597576  1.00000
  15       12       1     0.733333  0.114180  0.509545  0.95712
  22       11       1     0.666667  0.121716  0.428107  0.90523
  31       10       1     0.600000  0.126491  0.352082  0.84792
  35        9       1     0.533333  0.128812  0.280866  0.78580
  37        8       1     0.466667  0.128812  0.214199  0.71913
  39        7       1     0.400000  0.126491  0.152082  0.64792
  43        6       1     0.333333  0.121716  0.094774  0.57189
```

Distribution Analysis: NTime by NGroup

Comparison of Survival Curves

Test Statistics

Method	Chi-Square	DF	P-Value
Log-Rank	0.229974	1	0.632
Wilcoxon	0.073414	1	0.786

B.2 t-test

Two-sample T for Time

Group	N	Mean	StDev	SE Mean
OO	13	37.09	9.42	2.6
Proc	10	25.7	12.6	4.0

Difference = mu (D) - mu (E)
Estimate for difference: 11.39
95% CI for difference: (1.84, 20.94)
T-Test of difference = 0 (vs not =): T-Value = 2.48 P-Value = 0.022 DF = 21
Both use Pooled StDev = 10.9169

Appendix C

Materials for An Experiment Measuring the Effects of Activity on the Ability to Perform Maintenance

C.1 Introduction

This appendix contains all the materials used in the experiment described in chapter 6. I include the written specification of the system, the Enhancement and Documentation Initial tasks and the final Measured task. The original, rejected, Measured task and the alternate Measured tasks considered are also included at the end of this appendix.

C.2 Specification

C.2.1 Overview

The software is a command line interface to three different types of ranking systems. The three ranking systems are a ladder system, a league table and a points ranking system. The three systems are explained in more detail in the relevant sections C.2.4,C.2.5 and C.2.6. The command line parser is fairly primitive, commands are single keywords, which are case insensitive, followed by a fixed number of arguments depending on command, which are case sensitive. A list of valid commands and a further description of the operation of the command line is provided in section C.2.2. The ranking systems can be loaded from file and are automatically saved after every command that may modify them.

C.2.2 The Command Line Interface

As described the command line interface is primitive however it does have one interesting feature. Commands that are not appropriate for the current context (trying to LIST a ranking system when none is loaded) are not recognised. Any changes you make to the

software should keep this behaviour intact. Also whenever any commands that successfully modify the rankings within a system (i.e, ADD and RESULT) are executed, the ranking system is saved automatically.

Command List

Following are all the potential valid commands, when a command has arguments they are presented as <arg1><arg2>etc. There are never any optional arguments. If too many or too few arguments are provided then an appropriate error message will be displayed.

- QUIT - Exits the system.
- LOAD <fileName>- Load the given file, if it doesn't exist an error message will display, if the file doesn't contain a valid ranking system an error message will display.
- CREATE <fileName><rankingType>- Create a new empty ranking system which will have the given file name when saved. If this file currently exists it will be overwritten when the ranking system is saved. If rankingType isn't recognised an error message will be displayed. The three valid ranking systems names are `LadderSystem, LeagueSystem, PointsSystem`.
- HELP - List all the commands that are valid in this context
- ADD <name>- Add a new player with name to the ranking system, if the name already exists then an error message is displayed, if the League ranking system has started (see C.2.5) then an error message is displayed.
- LIST - Display the players in the ranking system ordered best-to-worst – top-to-bottom.
- [League System Loaded] RESULT <player1><player2><player1score><player2score>- Add in a result. If either player1 or player2 are not in the ranking system an error message is displayed. If either of the scores given is not a integer then an error message is displayed.
- [Ladder/Points System Loaded] RESULT <player1><player2><winner>- Add in a result that effects the ranking system - If either player1 or player2 are not in the ranking system then an error message is displayed. Winner is either the number of the winner (1 or 2) or 0 if the result is a tie. If winner is not 0, 1 or 2 an error message is displayed.
- [LeagueSystem Only] HISTORY <player1><player2>- Displays all the results of all games between player1 and player2, the order of player1 and player2 doesn't matter.

C.2.3 Error During Saving

The program recovers from all errors (that is displays and appropriate error message then continues) apart from one. If there is an error during the saving of a ranking system to

a file (which happens after every ADD and RESULT command) the prgroam displays an error message and quits.

C.2.4 Ladder System

A simple ranking system. When a player is added they are put in at the bottom of the ranking system. If players Al and Bob play against each other then one of three things can happen. If Al is below Bob and Al wins then he takes Bob's place and Bob moves down one place. If Al is above Bob and Al wins then nothing happens. If Al is above Bob and they draw then Bob is moved to the place immediately below Al.

File Format

```
LadderSystem
Al
Bob
Colin
```

The first line identifies the type of ranking system. The following lines are the players involved in the ladder with the person in first place being listed first.

C.2.5 League System

This is a implementation of a standard football style league system. Players are entered into the league with no points, as expected. Once the first result is added then no new players can be added to the league. Players get 3 points for a win, 1 point for a draw and no points for a loss. Players can only play each other a certain number of times (default 2, currently there is no way of changing that default). If a user tries to enter a extra result then an error message is displayed.

File Format

```
LeagueSystem
3
Al
Colin
Bob
2
Al Bob 4 1
Bob Colin 2 5
Al Colin 1 4
```

The first line identifies the type of ranking system. The first integer shows how many players are in the league. The players are then listed in no particular order. The next number is the maximum number of game that two players can play between each other. The remaining lines list all the games played between the players in the same format as results are enter on the CLI.

C.2.6 Points System

This ranking system is similar to the FIDE Chess ranking system. Each player added to this system is given 1000.0 points. If Al beats Bob then Bob gets a number of points deducted from their total and they are given to Al. The amount of points deducted/gained is based on the difference in scores between the two players. If Al and Bob have the same ranking and Al wins then he takes 50 points from Bob, if Bob had 200 points more than Al and Al won then Al would take 60 points off Bob. If Bob had 200 points less than Al and Al won then Al would take 40 points off Bob. In the event of a draw then the player with the smaller number of points is considered the winner, however they only get half as many points. When the two players have the same points total and it's a draw then there is no change to the points totals. The maximum points difference considered is 1000.0, so if Al had 3000.0 points and Bob had 500.0 points and Bob won the system would consider that a difference of 1000.0 points. Thus the maximum points available to win/lose is 100.0 points. If a player would lose enough points to give them a negative total the points deducted is changed to the amount that would reduce them to 0 points.

Comments

The Points system was written by someone who didn't understand how the program was structured, as a result we had to write a wrapper for it. On the positive side the class (`PointsRatings`) is partially commented.

File Format

```
PointsSystem
Al 1050.0
Bob 1000.0
Colin 950.0
```

The first line identifies the type of ranking system. The following lines are the players (and their scores) involved in the points system in numerically decreasing order.

C.2.7 Operation

Startup

Compile the code with `javac *.java`. Run the code with `java Shell`.

To run the code against the test suite use `java Shell hide <testSuite >output`. The correct output for the test suite is in the file `testOutput`.

Example Ranking System

Three Example ranking systems are provided in the files, `squash`, `tennis` and `chess` which correlate with `LadderSystem, LeagueSystem` and `PointsSystem`. To get a fresh version of them run the `cleanRankings.sh` script.

Code Location

The code can be found at `http://www.dcs.gla.ac.uk/~huttona/cleanCode.zip`

C.3 Initial Task - Enhancement

We would like you to add a single level `UNDO` feature to the system. Users should be able to take back the last `RESULT` command that was entered into the currently loaded ranking system. If the command executes successfully the message `Result Removed` should be displayed. If the user tries to `UNDO` multiple times in a row or tries to `UNDO` when no results have been added since a ranking system has been loaded the error `Nothing to UNDO` should be displayed. The `UNDO` command should be available whenever a ranking system is loaded. The response from the `HELP` command needs to be altered so that the line `UNDO - Takes back the last result entered into the system` is added.

C.3.1 Example

```
>Load ladder
File Loaded
>list
1)      Alistair
2)      John
3)      Bob
4)      Fred
>result Bob John 1
Result Added
>list
1)      Alistair
2)      Bob
3)      John
4)      Fred
>UNDO
Result Removed
>list
1)      Alistair
2)      John
3)      Bob
4)      Fred
>UNDO
Nothing to UNDO
```

C.3.2 Example Input and Output

The file `undoInput` contains a list of commands that checks to make sure the `UNDO` command is working. The file `undoOutput` is what should be produced by running `undoInput`.

C.4 Initial Task - Documentation

The majority of the program was written by a programmer who has now left. Unfortunately he failed to document the code at all. We need to make changes to the system however we are unwilling to do so without good documentation. As a result we would like you to add java doc commments to all the classes that make up the program. We would like you to produce a comment describing the purpose of each class first before moving on and providing comments for individual methods. For the class comments we would like you to to comment the classes you feel are most important first. Similarily for methods, start with what you think are the most important methods.

C.4.1 Example Comments

We would like you to produce cooments with a similar level of descriptiveness as the following.

```
/**
* This class represents a particle of an arbitary size
* in 2D space
*/
public class Particle
{
....

/**
* Determines if this particle will collide with another
* during a given amount of time and if it does at what time
* the collision will take place if the two particles keep
* moving in a straight line.  Resolves to Millisecond accuracy
*
* @see #collideTimeNano
*
* @param toTest The other particle which we are testing for collision
* @param time The length of time (in milliseconds) that will be checked over
*
* @return The time in millis of the collision otherwise -1 if they will not
* collide over the given time period.  This time is number of millis from now
*/
public long collideTimeMillis(Particle toTest, long time)

....
}
```

C.5 Final Measured Task

Currently the ranking systems offer three basic operations:

1. The ability to add a new player to the ranking system.
2. The ability to add aresult between two players in the rankign system.
3. The ability to get a listing of the players in the ranking system.

These are accessed using the commands `ADD`, `RESULT` and `LIST` respectively. We wish to add a fourth capability: The ability to determine if a player is already in the ranking system. We want this ability because individual ranking system have the the potential to become very large and as a result it would become hard to manually find if a player is in the ranking system. This new ability should be made accessible using the command `ISIN`. The specification for the command is:

`ISIN` <name>- Where name is the player's name that we wish to look-up in the currently loaded ranking system. If the name is in the ranking system the output of the command should be `Yes`. If the name is not in the ranking system the output of the command should be `No`. If no name is given or extra arguments are given then the standard error responses should be used.

This task can be split into two sub-tasks:

1. Alter the ranking systems so that they support the new ability.
2. Create the ISIN command and integrate it with the rest of the program.

We would like you to record the time that you finish each task. The input file `testSuite` can be used to test the correctness of your additions, the correct output is held in the file `testOutput`.

C.6 Example

```
>load squash
File loaded
>list
1)      Al
2)      Bob
3)      Colin
4)      Donald
5)      Edward
>isin Al
Yes
>isin Dond
No
>isin
Too feww arguments
>isin Al Donald
Too many arguments
```

C.7 Original Measured Task

We wish for the program to be changed so that it records a master rankings sytem. For this we assumme that all player names are unique (ie. that the Al reffered to in `LadderSystem` held in file `squash` is the same Al reffered to in the `PointsSystem` held in file `chess`. Overall ranking of players ability at playing games while individual ranking system relflect their ability at one particular activity. Every result and player that is added to a ranking system should also be added to this master ranking system. A new command (`MLIST`) is to be added that allows the Master ranking system to be listed. The Master ranking system is never itself `LOAD`ed up by the user, the results and players should be automatically added to the Master ranking system when they are succesfully added to the currently loaded ranking system. The Master ranking system is a `PointsSystem` PointsSystem stored in a file called `master`.

C.7.1 Sub-Tasks

This task is split into the following four sub-tasks:

1. Add code for master ranking system and implement `MLIST` so that the Master ranking system can be viewed.
2. Add the feature that when a player is added to the currently loaded ranking system they are added to the Master ranking system.
3. Add the feature that when a result is entered into the currently loaded ranking system is is added to the Master ranking system.
4. Add the `MLIST` command to the help system so that it appears when the `HELP` command is invoked.

Task 1 must be attempted first, when it is finished you may attempt tasks 2 and 3 in either order. Only once those are complete should you do the trivial task of adding `MILST` to the help system. Please record the times when you finish each of the sub-tasks. Further details on the subtasks are listed below.

MLIST

To allow the list to be viewed a new command should be added: `MLIST`. `MLIST` takes no arguments and all it does is list the master system like `LIST` displays the currently loaded system. `MLIST` is always available, that is it is available at the same level as `LOAD, CREATE, QUIT` and `HELP`.

Adding a Player

Whenever a player is succesfully added to the currently loaded system they should be added to the Master ranking system. If thier name already exists in the Master ranking system then suppress any error message produced by it ranking system.

Adding a Result

Whenever a result is succesfully added to the currently loaded system then the result should also be added to the Master ranking system.

Saving

Whenever a result or player is added to the Master ranking system then the Master ranking system should be saved just like a standard ranking system.

Loading

If the the user tries to load the master list then let them, this is just a prototoype so you don't have to worry about this case.

CREATE-ing a File Called Master

As with loading above do not consider the case when the users enter `Create master <RankingSystem>`.

Help

The response from the `HELP` command needs to be altered to reflect the new MLIST command. The help line should read `MLIST - Lists the Master Ranking System` and should appear on the line before the `CREATE` command.

C.7.2 Example Input and Output

A file called `sampleInput` containing a series of commands to test this new feature is provided. The output that the program should produce is shown in the file `expectedOutput`. There are also 4 files `task1, task2, task3, task4` which correlate with the 4 subtasks which test each feature individually. The correct output for these files is held in the files `out1, out2, out3, out4`. Remember to run `cleanRankings.sh` before running the program with the sample input.

C.7.3 Example of New System

```
>MLIST
Al 1050.0
Bob 1000.0
Colin 950.0
>Create sample LeagueSystem
Ranking System Created
>ADD Donald
Added: Donald
>LIST
Player          Played  Won     Drawn   Lost    Points
Donald          0       0       0       0       0
>MLIST
Al 1050.0
Bob 1000.0
Donald 1000.0
Colin 950.0

>Create another LadderSystem
Ranking System Created
>ADD Al
Added: Al
>ADD Donald
Added: Donald
>List
1)      Al
2)      Donald
>MList
Al 1050.0
Bob 1000.0
Donald 1000.0
Colin 950.0

>Result Donald Al
Result Added
>List
1)      Donald
2)      Al
>MLIST
Donald 1052.5
Bob 1000.0
Al 997.5
Colin 950.0
```

C.8 Alternate Measured Tasks

C.8.1 DIFF

We would like a new feature added to the three ranking systems. This feature will take two players that are in the rankings system and return the points (or place for the Ladder system) difference between the two players, the points/place difference returned is always positive. This new feature is to be accessible from the command line by using the command `DIFF`. The format of the `DIFF` command is as follows:

`DIFF <player1><player2>` where player1 and player2 are two players in the currently loaded system. If player1 or player2 do not exist in the system then the appropriate error message should be displayed. Otherwise the message `There is a differnece of <num>between <player1>and <player2>` where num is the difference between the two players.

The `DIFF` command should be available only when a ranking system is loaded. Its help text should be `DIFF - Display the points/place difference between two players`.

C.8.2 Case Sensitivity

Currently in the program commands are case insesitive but players' names are case sensitive. We would like the program altered so that for the purpose of entering results and using the `HISTORY` command players names are treated case insesitavely. Players' names will be stoed as there are given by the `ADD` command however their case will be ignored when using other commands. For example if the command `ADD Donald` was executed the name would appear as `Donald` when the ranking system was listed but the command `RESULT DoNaLd Al 1` would execute succesfully. A side effect of this change would mean that you couldn't `ADD DONALD` to a system that already contained `Donald`.

C.9 Code

This section contains all the code necessary for performing the experiment.

```
public class AddPlayerCommand extends Command
{

    String name;

    public AddPlayerCommand()
    {
        super();
    }

    public AddPlayerCommand(String name, RankingSystemI rs)
    {
        super(rs);
        this.name = name;
    }

    public String[] execute()
    {
        String[] output = new String[1];
        try
        {
            rs.addPlayer(name);
            output[0] = "Added: " + name;
            RankingSystemLoader.saveFile(rs);
        }
        catch (NameAlreadyExists e)
        {
            output[0] = name + " already exists in ranking system";
        }
        return output;
    }

}
```

```
public class AlreadyStarted extends Exception
{

    public AlreadyStarted()
    {
        super();
    }

    public AlreadyStarted(String s)
    {
        super(s);
    }

}
```

```
import java.util.StringTokenizer;

public class BasicCommandFactory extends CommandFactory
{

    public BasicCommandFactory(RankingSystemI rs,Shell sh)
    {
```

```
8           super(rs,sh);
9       }
10
11      public Command generateCommand(String command, String args,
12                                      CommandFactory topLevel) throws
13                                          UnknownCommand,
14                                          IncorrectArguments,
15                                          NameDoesntExist
16      {
17          StringTokenizer st = new StringTokenizer (args);
18          if (command.equals("QUIT"))
19          {
20              if (st.hasMoreTokens()) return new ErrorCommand (new
21                      IncorrectArguments(IncorrectArguments.TOOMANY));
22              return new QuitCommand();
23          }
24          else if (command.equals("HELP"))
25          {
26              if (st.hasMoreTokens()) return new ErrorCommand (new
27                      IncorrectArguments(IncorrectArguments.TOOMANY));
28              return new HelpCommand(topLevel);
29          }
30          else if (command.equals("LOAD"))
31          {
32              if (!st.hasMoreTokens()) return new ErrorCommand (new
33                      IncorrectArguments(IncorrectArguments.TOOFEW));
34              String arg = st.nextToken();
35              if (st.hasMoreTokens()) return new ErrorCommand (new
36                      IncorrectArguments(IncorrectArguments.TOOMANY));
37              return new LoadCommand(rs,arg,this);
38
39          }
40          else if (command.equals("CREATE"))
41          {
42                  if (!st.hasMoreTokens()) return new ErrorCommand (new
43                          IncorrectArguments(IncorrectArguments.TOOFEW));
44                  String name = st.nextToken();
45                          if (!st.hasMoreTokens()) return new ErrorCommand (new
46                                  IncorrectArguments(IncorrectArguments.TOOFEW));
47                          String type = st.nextToken();
48                          if (st.hasMoreTokens()) return new ErrorCommand (new
49                                  IncorrectArguments(IncorrectArguments.TOOMANY));
50                          return new CreateCommand(rs, sh ,name,type);
51          }
52          else
53          {
54              return super.generateCommand(command,args,topLevel);
55          }
56
57      }
58
59      public String[] commandList(String[] otherCommands)
60      {
61          int length = otherCommands.length + 4;
62          String[] coms = new String[length];
63          coms[0] = "QUIT - Exits the Shell Program";
64          coms[1] = "HELP - Lists all available Commands";
65          coms[2] = "LOAD - Loads up the named ranking file";
66          coms[3] = "CREATE - Create a new empty ranking system";
67          System.arraycopy(otherCommands,0,coms,4,length-4);
68
```

```
        return  super.commandList(coms);
    }
}
```

```
import java.util.StringTokenizer;

public class BasicListCommandFactory extends BasicCommandFactory
{
    public BasicListCommandFactory(RankingSystemI rs, Shell sh)
    {
        super(rs,sh);
    }

    public Command generateCommand(String command, String args,
                                   CommandFactory topLevel) throws
                                   UnknownCommand,
                                   IncorrectArguments,
                                   NumberFormatException,
                                   NameDoesntExist
    {
        StringTokenizer st = new StringTokenizer(args);
        if (command.equals("LIST"))
        {
            if (st.hasMoreTokens()) return new ErrorCommand (new
                        IncorrectArguments(IncorrectArguments.TOOMANY));
            return new ListCommand(rs);
        }
        else if (command.equals("ADD"))
        {
            String name;
            if (!st.hasMoreTokens()) return new ErrorCommand (new
                        IncorrectArguments(IncorrectArguments.TOOFEW));
            else name = st.nextToken();
            if (st.hasMoreTokens()) return new ErrorCommand (new
                        IncorrectArguments(IncorrectArguments.TOOMANY));

            return new AddPlayerCommand(name,rs);
        }
        else if (command.equals("RESULT"))
        {
            String p1;
            String p2;
            int result;

            if (st.hasMoreTokens())
            {
                p1 = st.nextToken();
            }
            else
            {
                return new ErrorCommand (new
                            IncorrectArguments(IncorrectArguments.TOOFEW));
            }

            if (st.hasMoreTokens())
            {
                p2 = st.nextToken();
            }
            else
            {
                return new ErrorCommand (new
```

```
                            IncorrectArguments(IncorrectArguments.TOOFEW));
            }

            if (st.hasMoreTokens())
            {
                String res = st.nextToken();
                try
                {
                    result = Integer.parseInt(res);
                    if (result != 0 && result != 1 && result != 2) return
                    new ErrorCommand(new
                        IncorrectArguments(result + " should be a 0, 1 or 2"));
                }
                catch (NumberFormatException nfe)
                {
                    return new ErrorCommand(new
                            IncorrectArguments(res + " should be a number"));
                }
            }
            else
            {
                return new ErrorCommand (new
                                IncorrectArguments(IncorrectArguments.TOOFEW));
            }
            if (st.hasMoreTokens()) return new ErrorCommand (new
                               IncorrectArguments(IncorrectArguments.TOOMANY));
            try
            {
                return new ResultCommand(p1, p2, result, rs);
            }
            catch (SameNameException e)
            {
                return new ErrorCommand(e);
            }
        }
        else
        {
            return super.generateCommand(command,args,topLevel);
        }
    }

    public String[] commandList(String[] otherCommands)
    {
        int length = otherCommands.length + 3;
        String[] coms = new String[length];
        coms[0] = "RESULT - Add a result into the system";
        coms[1] = "ADD - Add an additional player to the ranking system";
        coms[2] = "LIST - Display the public listing of the ranking system";
        System.arraycopy(otherCommands,0,coms,3,length-3);

        return  super.commandList(coms);
    }
}
```

```
public class BasicResult implements ResultI
{

    public static int P1BEATP2 = 1;
    public static int P2BEATP1 = 2;
    public static int DRAW = 0;

```

```
    public String player1;
    public String player2;
    public int result;

    BasicResult()
    {
    }

    BasicResult(String player1, String player2, int result)
    {
        this.player1 = player1;
        this.player2 = player2;
        this.result = result;
    }
}
```

```
public abstract class CommandFactory
{
    protected RankingSystemI rs;
    protected Shell sh;

    public CommandFactory(RankingSystemI rs, Shell sh)
    {
        this.rs = rs;
        this.sh = sh;
    }

    public Command generateCommand(String command, String args,
                                   CommandFactory topLevel) throws
                                                    UnknownCommand,
                                                    IncorrectArguments,
                                                    NameDoesntExist
    {
        return new ErrorCommand(new UnknownCommand(command));
    }

    public String[] commandList(String[] otherCommands)
    {
        return otherCommands;
    }
}
```

```
public abstract class Command
{
    RankingSystemI rs;

    Command()
    {
        this.rs = null;
    }

    Command(RankingSystemI rs)
    {
        this.rs = rs;
    }

    public abstract String[] execute();
}
```

```
public class CreateCommand extends Command
{
    String fileName;
    String rsType;
    Shell sh;

    public CreateCommand(RankingSystemI rs, Shell sh, String fileName,
                                                                String rsType)
    {
        super(rs);
        this.fileName = fileName;
        this.rsType = rsType;
        this.sh = sh;
    }

    public String[] execute()
    {
        String[] output = {"Ranking System Created"};
        RankingSystemLoader.newFile(fileName);
        if (rsType.equals("LadderSystem"))
        {
            new LadderSystem(sh);
        }
        else if (rsType.equals("LeagueSystem"))
        {
            new LeagueSystem(sh);
        }
        else if (rsType.equals("PointsSystem"))
        {
            new PointsShell(sh);
        }
        else
        {
            output[0] = "Unrecognised Ranking System";
        }

        return output;
    }
}
```

```
public class ErrorCommand extends Command
{
    Exception e;

    public ErrorCommand(Exception e)
    {
        this.e = e;
    }

    public String[] execute()
    {
        String[] output = new String[1];
        output[0] = e.toString();
        return output;
    }
}
```

```
public class HelpCommand extends Command
{
```

```
3
4       CommandFactory cf;
5
6       public HelpCommand()
7       {
8           super();
9       }
10
11      public HelpCommand(CommandFactory cf)
12      {
13          this.cf = cf;
14      }
15
16      public String[] execute()
17      {
18          String[] commands = {};
19          return cf.commandList(commands);
20      }
21
22  }
```

```
1   import java.util.LinkedList;
2   import java.util.ListIterator;
3
4   public class HistoryCommand extends Command
5   {
6
7       String player1;
8       String player2;
9
10      public HistoryCommand(String player1, String player2, RankingSystemI rs)
11      {
12          super(rs);
13          this.player1 = player1;
14          this.player2 = player2;
15      }
16
17      public String[] execute()
18      {
19          String[] output = null;
20          LinkedList llout = new LinkedList();
21          LinkedList results = ((LeagueSystem) rs).results;
22          ListIterator li = results.listIterator();
23          while(li.hasNext())
24          {
25              LeagueSystem.LeagueResult lr = (LeagueSystem.LeagueResult) li.next();
26              if ((lr.player1.equals(player1) || lr.player1.equals(player2)) &&
27                  (lr.player2.equals(player1) || lr.player2.equals(player2)))
28              {
29                  llout.add(lr.toString());
30              }
31          }
32
33          output = new String[llout.size()];
34          for(int i = 0; i < output.length; i ++)
35          {
36              output[i] = (String) llout.get(i);
37          }
38          return output;
39      }
40
```

```
}
```

```
public class IncorrectArguments extends Exception
{
    public static String TOOMANY = "Too many arguments";
    public static String TOOFEW = "Too feww arguments";
    String s;

    public IncorrectArguments(String s)
    {
        this.s = s;
    }

    public String toString()
    {
        return s;
    }
}
```

```
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.ListIterator;

public class LadderSystem implements RankingSystemI
{

    public class LadderCommands extends BasicListCommandFactory
    {
        public LadderCommands(RankingSystemI rs,Shell sh)
        {
            super(rs,sh);
        }
    }

    public class LadderPerson
    {
        String name;
        LadderPerson(String name)
        {
            this.name = name;
        }

        public String toString()
        {
            return name;
        }
    }

    protected ArrayList players = new ArrayList();

    public LadderSystem(Shell sh)
    {
        sh.switchCommandSource(new LadderCommands(this,sh));
    }

    public LadderSystem(Shell sh, LinkedList ll)
    {
        ListIterator li = ll.listIterator();
```

```
42          while (li.hasNext())
43          {
44              players.add(new LadderPerson((String)li.next()));;
45          }
46
47          sh.switchCommandSource(new LadderCommands(this,sh));
48      }
49
50      public void addPlayer(String name) throws NameAlreadyExists
51      {
52          int index = find(name);
53          if (index == -1)
54              players.add(new LadderPerson(name));
55          else
56              throw new NameAlreadyExists(name);
57      }
58
59      public void addResult(ResultI result) throws NameDoesntExist,
60                                                   IncorrectArguments
61      {
62          BasicResult r = (BasicResult) result;
63
64          int p1 = find(r.player1);
65          int p2 = find(r.player2);
66
67          if (p1 == -1) throw new NameDoesntExist(r.player1);
68          if (p2 == -1) throw new NameDoesntExist(r.player2);
69          if (!(r.result == BasicResult.P1BEATP2 |
70                  r.result == BasicResult.P2BEATP1 |
71                  r.result == BasicResult.DRAW ))
72              throw new IncorrectArguments(r.result +
73                              " is an unrecognised result code");
74
75          if (r.result == BasicResult.P1BEATP2 && p1 > p2)
76          {
77              adjust(p2,p1);
78          }
79          else if (r.result == BasicResult.P2BEATP1 && p2 > p1)
80          {
81              adjust(p1,p2);
82          }
83      }
84
85      private void adjust(int down, int up)
86      {
87          Object k;
88          k = players.get(down);
89          players.set(down,players.get(up));
90          players.remove(up);
91          players.add(down+1,k);
92      }
93
94      private int find(String name)
95      {
96          int player = -1;
97          for (int i = 0; i < players.size(); i ++)
98          {
99              if (name.equals(((LadderPerson)players.get(i)).name))
100             {
101                 player = i;
102                 break;
```

```
            }
        }
        return player;
    }

    public String[] publicListing()
    {
        String[] output = new String[players.size()];
        ListIterator li = players.listIterator();
        int counter = 0;
        while(li.hasNext())
        {
            output[counter] = (counter + 1) + ")\t" +
                        ((LadderPerson)li.next()).toString();
            counter ++;
        }

        return output;
    }

    public void save(FileWriter writer) throws IOException
    {
        writer.write("LadderSystem\n");
        for (int i = 0; i < players.size(); i ++)
        {
            writer.write(((LadderPerson)players.get(i)).name + "\n");
        }
    }




}
```

```
import java.io.FileWriter;
import java.io.IOException;
import java.util.LinkedList;
import java.util.ListIterator;
import java.util.NoSuchElementException;
import java.util.StringTokenizer;

public class LeagueSystem implements RankingSystemI
{

    public class LeagueAddPlayerCommand extends AddPlayerCommand
    {
        public LeagueAddPlayerCommand(String name, RankingSystemI rs)
        {
            super(name,rs);
        }

        public String[] execute()
        {
            String[] output = new String[1];
            if (((LeagueSystem)rs).hasStarted()) output[0] =
                    "League has started new players cannot be added";
            else output = super.execute();

            return output;
        }
    }
```

```
28  public class LeagueCommandFactory extends BasicListCommandFactory
29  {
30      public LeagueCommandFactory(RankingSystemI rs, Shell sh)
31      {
32          super(rs, sh);
33      }
34
35      public String[] commandList(String[] otherCommands)
36      {
37          int length = otherCommands.length + 1;
38          String[] coms = new String[length];
39          coms[0] = "HISTORY - Shows the results between two players";
40          System.arraycopy(otherCommands,0,coms,1,length-1);
41
42          return  super.commandList(coms);
43      }
44
45      public Command generateCommand(String command, String args,
46                                              CommandFactory topLevel) throws
47                                                          UnknownCommand,
48                                                          IncorrectArguments,
49                                                          NameDoesntExist
50      {
51          StringTokenizer st = new StringTokenizer(args);
52          if (command.equals("RESULT"))
53          {
54              int p1Score = 0, p2Score = 0;
55              String score = null;
56              try
57              {
58                  String player1 = st.nextToken();
59                  String player2 = st.nextToken();
60                  score = st.nextToken();
61                  p1Score = Integer.parseInt(score);
62                  score = st.nextToken();
63                  p2Score = Integer.parseInt(score);
64                  if (st.hasMoreTokens()) return new ErrorCommand (new
65                              IncorrectArguments(IncorrectArguments.TOOMANY));
66                  return new LeagueResultCommand(player1,player2,p1Score,
67                                                                  p2Score,rs);
68              }
69              catch (SameNameException e)
70              {
71                  return new ErrorCommand(e);
72              }
73              catch (NoSuchElementException e)
74              {
75                  return new ErrorCommand (new
76                            IncorrectArguments(IncorrectArguments.TOOFEW));
77              }
78              catch (NumberFormatException e)
79              {
80                  return new ErrorCommand(new
81                            IncorrectArguments(score + " should be a number"));
82              }
83          }
84          else if (command.equals("ADD"))
85          {
86              String name;
87              if (!st.hasMoreTokens()) return new ErrorCommand (new
88                               IncorrectArguments(IncorrectArguments.TOOFEW));
```

```
89                  else name = st.nextToken();
90                  if (st.hasMoreTokens()) return new ErrorCommand (new
91                                  IncorrectArguments(IncorrectArguments.TOOMANY));
92
93                  return new LeagueAddPlayerCommand(name,rs);
94              }
95              else if (command.equals("HISTORY"))
96              {
97                  try
98                  {
99                      String player1 = st.nextToken();
100                     String player2 = st.nextToken();
101
102                     if (((LeagueSystem)rs).find(player1)  == -1)
103                     {
104                         return new ErrorCommand(new NameDoesntExist(player1));
105                     }
106                     if (((LeagueSystem)rs).find(player2)  == -1)
107                     {
108                         return new ErrorCommand(new NameDoesntExist(player2));
109                     }
110                     return new HistoryCommand(player1,player2,rs);
111                 }
112                 catch (NoSuchElementException e)
113                 {
114                     return new ErrorCommand(new
115                                 IncorrectArguments(IncorrectArguments.TOOFEW));
116                 }
117             }
118             else
119             {
120                 return super.generateCommand(command,args,topLevel);
121             }
122         }
123     }
124
125     class LeaguePerson
126     {
127         int drawn = 0;
128         int lost = 0;
129         String name;
130         int points = 0;
131         LinkedList results = new LinkedList();
132         int won = 0;
133
134         LeaguePerson (String name)
135         {
136             this.name = name;
137         }
138
139         public String toString()
140         {
141             String firstSpace = "\t";
142             if (name.length() < 8) firstSpace = "\t\t";
143
144             return name + firstSpace + (won+lost+drawn)+ "\t"+won+ "\t"+ drawn+
145                     "\t" +  lost + "\t"+points;
146         }
147     }
148
149     public class LeagueResult extends BasicResult
```

```
150     {
151         int p1Score;
152         int p2Score;
153 
154         public LeagueResult(String storedForm)
155         {
156             StringTokenizer st = new StringTokenizer(storedForm);
157             player1 = st.nextToken();
158             player2 = st.nextToken();
159             p1Score = Integer.parseInt(st.nextToken());
160             p2Score = Integer.parseInt(st.nextToken());
161             if (p1Score > p2Score) result = BasicResult.P1BEATP2;
162             else if (p1Score < p2Score) result = BasicResult.P2BEATP1;
163             else result = BasicResult.DRAW;
164         }
165 
166         public LeagueResult(String player1, String player2, int p1Score,
167                                                                 int p2Score)
168         {
169             super(player1,player2,99);
170             this.p1Score = p1Score;
171             this.p2Score = p2Score;
172             if (p1Score > p2Score) result = BasicResult.P1BEATP2;
173             else if (p1Score < p2Score) result = BasicResult.P2BEATP1;
174             else result = BasicResult.DRAW;
175         }
176 
177         public String toString()
178         {
179             return player1 + " " + player2 + " " + p1Score + " " + p2Score;
180         }
181     }
182 
183     public class LeagueResultCommand extends ResultCommand
184     {
185         public LeagueResultCommand(String p1,
186                                    String p2,
187                                    int p1Score,
188                                    int p2Score,
189                                    RankingSystemI rs) throws SameNameException
190         {
191             if (p1.equals(p2)) throw new SameNameException();
192             this.r = new LeagueResult(p1,p2,p1Score,p2Score);
193             this.rs = rs;
194         }
195 
196         public String[] execute()
197         {
198             String[] history;
199             String[] output = new String[1];
200             HistoryCommand hc = new HistoryCommand(((LeagueResult)r).player1,
201                                                    ((LeagueResult)r).player2,rs);
202             history = hc.execute();
203             if (history.length == numGames) output[0] =
204                 "Players have already faced each other maximum number of times";
205             else output = super.execute();
206             return output;
207         }
208     }
209 
210     private static int DRAW = 1;
```

```
211     private static int LOSE = 0;
212     private static int WIN = 3;
213 
214     protected int numGames = 2;
215 
216     protected LinkedList players = new LinkedList();
217     protected LinkedList results = new LinkedList();
218     protected boolean started = false;
219 
220     public LeagueSystem(Shell sh)
221     {
222         super();
223         sh.switchCommandSource(new LeagueCommandFactory(this,sh));
224     }
225 
226     public LeagueSystem(Shell sh, LinkedList ll)
227     {
228         ListIterator li = ll.listIterator();
229         int teams = Integer.parseInt((String)li.next());
230         for(int i = 0 ; i < teams; i ++)
231         {
232             players.add(new LeaguePerson((String)li.next()));;
233         }
234 
235         numGames = Integer.parseInt((String)li.next());
236 
237         while(li.hasNext())
238         {
239             String line = (String) li.next();
240             try {
241             addResult(new LeagueResult(line));
242             } catch (Exception e) {}
243         }
244 
245         sh.switchCommandSource(new LeagueCommandFactory(this,sh));
246     }
247 
248     public void addPlayer(String name) throws NameAlreadyExists
249     {
250         if (find(name) != -1) throw new NameAlreadyExists(name);
251         players.add(new LeaguePerson(name));
252     }
253 
254     public void addResult(ResultI r) throws NameDoesntExist
255     {
256         LeagueResult result = (LeagueResult) r;
257         int p1 = find(result.player1);
258         int p2 = find(result.player2);
259         if (p1 == -1) throw new NameDoesntExist(result.player1);
260         if (p2 == -1) throw new NameDoesntExist(result.player2);
261         started = true;
262         results.add(result);
263 
264         LeaguePerson pl1 = (LeaguePerson)players.get(p1);
265         LeaguePerson pl2 = (LeaguePerson)players.get(p2);
266 
267         if (result.result == BasicResult.P1BEATP2)
268         {
269             pl1.points += WIN;
270             pl1.won += 1;
271             pl2.points += LOSE;
```

```
272             pl2.lost += 1;
273         }
274         else if (result.result == BasicResult.P2BEATP1)
275         {
276             pl1.points += LOSE;
277             pl1.lost += 1;
278             pl2.points += WIN;
279             pl2.won += 1;
280         }
281         else
282         {
283             pl1.points += DRAW;
284             pl1.drawn += 1;
285             pl2.points += DRAW;
286             pl2.drawn += 1;
287         }
288     }
289 
290     private int find(String name)
291     {
292         int player = -1;
293         for (int i = 0; i < players.size(); i ++)
294         {
295             if (name.equals(((LeaguePerson)players.get(i)).name))
296             {
297                 player = i;
298                 break;
299             }
300         }
301         return player;
302     }
303 
304     public boolean hasStarted()
305     {
306         return started;
307     }
308 
309     public String[] publicListing()
310     {
311         LeaguePerson[] people = new LeaguePerson[players.size()];
312         for(int i = 0; i < players.size(); i ++)
313         {
314             people[i] = (LeaguePerson) players.get(i);
315         }
316 
317         for(int i = 0; i < players.size()-1; i ++)
318         {
319             for(int j = i+1; j < players.size(); j ++)
320             {
321                 if (people[i].points < people[j].points)
322                 {
323                     LeaguePerson temp = people[i];
324                     people[i] = people[j];
325                     people[j] = temp;
326                 }
327             }
328         }
329 
330         String[] output = new String[people.length+1];
331         output[0] = "Player\t\tPlayed\tWon\tDrawn\tLost\tPoints";
332         for(int i = 0; i < people.length; i ++)
```

```
        {
            output[i+1] = people[i].toString();
        }
        return output;
    }

    public void save(FileWriter writer) throws IOException
    {
        writer.write("LeagueSystem\n");
        writer.write(players.size()+"\n");

        ListIterator li = players.listIterator();
        while(li.hasNext())
        {
            LeaguePerson lp = (LeaguePerson) li.next();
            writer.write(lp.name+"\n");
        }
        writer.write(""+numGames+"\n");
        li = results.listIterator();
        while(li.hasNext())
        {
            LeagueResult lr = (LeagueResult) li.next();
            writer.write(lr.toString() + "\n");
        }
    }

}
```

```
public class ListCommand extends Command
{
    public ListCommand()
    {
        super();
    }

    public ListCommand(RankingSystemI rs)
    {
        super(rs);
    }

    public String[] execute()
    {
        return rs.publicListing();
    }

}
```

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.LinkedList;

public class LoadCommand extends Command
{

    String fileName;
    CommandFactory cf;

    public LoadCommand()
    {
        super();
```

```
14      }
15
16      public LoadCommand(RankingSystemI rs, String fileName, CommandFactory cf)
17      {
18          super(rs);
19          this.fileName = fileName;
20          this.cf = cf;
21      }
22
23      private RankingSystemI decideSystem(String type, LinkedList input)
24      {
25          if (type.equals("LadderSystem"))
26          {
27              return new LadderSystem(cf.sh,input);
28          }
29          else if (type.equals("LeagueSystem"))
30          {
31              return new LeagueSystem(cf.sh,input);
32          }
33          else if (type.equals("PointsSystem"))
34          {
35              return new PointsShell(cf.sh,input);
36          }
37          else
38          {
39              return null;
40          }
41      }
42
43      public String[] execute()
44      {
45          String[] output = {"File Loaded"};
46          try
47          {
48              Object[] o = RankingSystemLoader.loadFile(fileName);
49
50              LinkedList fileInput = (LinkedList)o[1];
51              String type = (String)o[0];
52
53              RankingSystemI newRankingSystem = decideSystem(type,fileInput);
54              if (newRankingSystem == null)
55              {
56                  output[0] = "Ranking System Not Recognised";
57                  return output;
58              }
59          }
60          catch (FileNotFoundException e)
61          {
62              output[0] = "File does not exist";
63          }
64          catch (IOException e)
65          {
66              output[0] = "Problem loading file";
67          }
68
69          return output;
70      }
71
72  }
```

```
public class NameAlreadyExists extends Exception
{
    String name;

    NameAlreadyExists(String name)
    {
       this.name = name;
    }

    public String toString()
    {
       String str = "The name: " + name + " alreasy exists in this ranking file";
       return str;
    }
}
```

```
public class NameDoesntExist extends Exception
{
    private String name;

    public NameDoesntExist(String name)
    {
        this.name = name;
    }

    public String toString()
    {
        return "The name "+name+" does not exist";
    }
}
```

```
import java.util.Collections;
import java.util.Vector;

public class PointsRatings
{

    public class Person implements Comparable
    {
        public double points;
        public String name;
        Person(String n, double p)
        {
            name = n;
            points = p;
        }

        public int compareTo(Object o)
        {
            if (((Person)o).points < this.points)
            {
                return -1;
            }
            else if (((Person)o).points > this.points)
            {
                return 1;
            }
            else
            {
```

```
29                  return 0;
30              }
31          }
32      }
33
34      Vector v = new Vector();
35
36      public PointsRatings()
37      {
38      }
39
40      /**
41       * Add a new person into the ratings with the given name and starting points,
42       * if ther is already a person with that name then an error
43       * code is returned otherwise 1 is returned
44       *
45       * @param name The naem of the new person
46       * @param points How many points they should start with
47       * @return int The error code for duplicate name is -1;
48       */
49      public int newPerson(String name, double points)
50      {
51          if (getPerson(name) != null) return -1;
52          v.add(new Person(name,points));
53          return 1;
54      }
55
56
57      /**
58       * Lookup the name given in the ratings, if the name
59       * matches a Person in the ratings then return that
60       * person otherwise return null
61       *
62       * @param name The name to check
63       * @return Person Is null if the name is not in the ratings
64       */
65      public Person getPerson(String name)
66      {
67          for (int i =0; i < v.size(); i ++)
68          {
69              if (((Person)v.get(i)).name.equals(name))
70              {
71                  return (Person)v.get(i);
72              }
73          }
74
75          return null;
76      }
77
78      /**
79       * Return the person at position i in the ratings
80       *
81       * @param i The Person to return
82       * @return Person
83       */
84      public Person getPerson(int i)
85      {
86          return (Person)v.get(i);
87      }
88
89      /**
```

```
90      * Change the players scores the the appropriate amount.
91      * Max points diff consdiered is 1000 points.  PLayers
92      * lose/gain 100 to 0 points.  In the event of a draw
93      * the player with the least number of points is considered
94      * the winner but they get half the points they would have.
95      *
96      * @param winner The player who won
97      * @param loser The player who lost
98      * @param draw Wether the game was a draw
99      */
100     public void calculateScore(Person winner, Person loser, boolean draw)
101     {
102         if (draw && winner.points > loser.points)
103         {
104             Person temp = winner;
105             winner = loser;
106             loser = temp;
107         }
108         else if (draw && winner.points == loser.points)
109         {
110             return;
111         }
112         double diff = loser.points - winner.points;
113         diff += 1000;
114         if (diff < 0) diff = 0;
115         if (diff > 2000) diff = 2000;
116         double max = 100;
117 
118         double score = max *(diff/2000);
119         if (draw) score /= 2;
120         if (loser.points - score < 0) score = loser.points;
121         winner.points += score;
122         loser.points -= score;
123     }
124 
125     /**
126      * Add in a result to this Rating.
127      *
128      * @param a A valid Person in the Rating
129      * @param b A valid Person in the Rating
130      * @param result Should 1 if a is the winner, -1 if b is
131      * the winner and anything else if it's a draw.
132      */
133     public void result(Person a,Person b, int result)
134     {
135         if (result == 1)
136         {
137             calculateScore(a,b,false);
138         }
139         else if (result == -1)
140         {
141             calculateScore(b,a,false);
142         }
143         else
144         {
145             calculateScore(a,b,true);
146         }
147     }
148 
149     /**
150      *
```

```
     *
     * @see java.lang.Object#toString()
     */
    public String toString()
    {
        Collections.sort(v);
        String str = "";
        for (int i = 0; i < v.size(); i ++)
        {
            str += getPerson(i).name + " " + getPerson(i).points+"\n";
        }
        return str;
    }

    /**
     * Test Harness
     * @param args
     */
    public static void main(String[] args)
    {
        PointsRatings pr = new PointsRatings();
        pr.newPerson("A",1000.0);
        pr.newPerson("B",1000.0);

        pr.result(pr.getPerson("A"),pr.getPerson("B"),1);
        System.out.println(pr);
        pr.newPerson("C",1000.0);
        System.out.println(pr);
    }
}
```

```
import java.io.FileWriter;
import java.io.IOException;
import java.util.LinkedList;
import java.util.ListIterator;

public class PointsShell implements RankingSystemI
{
    PointsRatings pr;
    Shell sh;

    public class PointsCommandFactory extends BasicListCommandFactory
    {
        public PointsCommandFactory(RankingSystemI rs, Shell sh)
        {
            super(rs,sh);
        }
    }

    public PointsShell(Shell sh, LinkedList ll)
    {
        this.sh = sh;
        pr = new PointsRatings();

        ListIterator li = ll.listIterator();
        while (li.hasNext())
        {
            String line = (String)li.next();
            String name = line.substring(0,line.indexOf(" "));
            double points = Double.parseDouble(
                                line.substring(line.indexOf(" ")+1));
```

```
            pr.newPerson(name,points);
        }
        sh.switchCommandSource(new PointsCommandFactory(this,sh))    ;
    }

    public PointsShell(Shell sh)
    {
        this.sh = sh;
        pr = new PointsRatings();
        sh.switchCommandSource(new PointsCommandFactory(this,sh));
    }

    public void addPlayer(String name) throws NameAlreadyExists
    {
        PointsRatings.Person person = pr.getPerson(name);
        if (person != null) throw new NameAlreadyExists(name);
        else pr.newPerson(name,1000.0);
    }

    public void addResult(ResultI r) throws NameDoesntExist, IncorrectArguments
    {
        BasicResult br = (BasicResult) r;

        PointsRatings.Person p1 = pr.getPerson(br.player1);
        PointsRatings.Person p2 = pr.getPerson(br.player2);
        if (p1 == null)  throw new NameDoesntExist(br.player1);
        if (p2 == null)  throw new NameDoesntExist(br.player2);

        if (br.result == 1)
        {
            pr.result(p1,p2,1);
        }
        else if (br.result == 2)
        {
            pr.result(p1,p2,-1);
        }
        else
        {
            pr.result(p1,p2,0);
        }
    }

    public String[] publicListing()
    {
        String[] str = new String[1];
        str[0] = pr.toString();
        return str;
    }

    public void save(FileWriter writer) throws IOException
    {
        writer.write("PointsSystem\n");
        writer.write(pr.toString());
        writer.flush();
    }

}
```

```
public class QuitCommand extends Command
{
    public QuitCommand()
```

```
    {
        super();
    }

    public String[] execute()
    {
        System.exit(0);
        return null;
    }
}
```

```
import java.io.FileWriter;
import java.io.IOException;

public interface RankingSystemI
{
    public static int ASCENDING = 0;
    public static int DESCENDING = 1;

    public void addPlayer(String name) throws NameAlreadyExists;

    public void addResult(ResultI r) throws NameDoesntExist, IncorrectArguments;


    public String[] publicListing();

    public void save(FileWriter writer) throws IOException;
}
```

```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.LineNumberReader;
import java.util.LinkedList;

public abstract class RankingSystemLoader
{

    static File rankings;
    static File oldFile;

    public RankingSystemLoader()
    {
        rankings = null;
        oldFile = null;
    }

    public static Object[] loadFile(String fileName) throws IOException
    {
        oldFile = rankings;

        Object[] o = new Object[2];
        try
        {
            rankings = new File(fileName);

            LineNumberReader in = new LineNumberReader(new FileReader(rankings));
            String type = in.readLine();
            LinkedList ll = new LinkedList();
```

```
            String input = in.readLine();
            while (input != null)
            {
                ll.add(input);
                input = in.readLine();
            }

            o[0] = type;
            o[1] = ll;
        }
        catch (IOException e)
        {
            rankings = oldFile;
            throw e;
        }
        return o;
    }

    public static void newFile(String fileName)
    {
        rankings = new File(fileName);
    }

    public static void saveFile(RankingSystemI rsi)
    {
        try
        {
            FileWriter out = new FileWriter(rankings);
            rsi.save(out);
            out.close();
        }
        catch (Exception e)
        {
            System.err.println("Critical Error saving file");
            System.exit(-2);
        }
    }
}
```

```
public class ResultCommand extends Command
{
    ResultI r;
    RankingSystemI rs;

    public ResultCommand()
    {
    }

    public ResultCommand(String p1, String p2, int result, RankingSystemI rs)
                                                            throws SameNameException
    {
        if (p1.equals(p2)) throw new SameNameException();
        this.r = new BasicResult(p1,p2,result);
        this.rs = rs;
    }

    public String[] execute()
    {
        String str[] = {"Result Added"};
        try
        {
```

```
            rs.addResult(r);
            RankingSystemLoader.saveFile(rs);
        }
        catch (NameDoesntExist e)
        {
            str[0] = e.toString();
        }
        catch (IncorrectArguments e)
        {
            str[0] = e.toString();
        }

        return str;
    }
}
```

```
public interface ResultI
{

}
```

```
public class SameNameException extends Exception
{
    public String toString()
    {
        return "Players cannot play themselves";
    }
}
```

```
public class SaveCommand extends Command
{
    public SaveCommand()
    {
        super();
    }

    public SaveCommand(RankingSystemI rs)
    {
        super(rs);
    }

    public String[] execute()
    {
        String[] output =  {"File Saved"};
        try
        {
            RankingSystemLoader.saveFile(rs);
            return output;
        }
        catch (Exception e)
        {
            output[0] = "File not Saved";
            return output;
        }
    }
}
```

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
```

```
3  import java.io.IOException;
4  import java.io.InputStreamReader;
5  import java.io.OutputStreamWriter;
6
7  public class Shell
8  {
9      protected BufferedReader input;
10     protected BufferedWriter output;
11     protected String prompt = ">";
12     protected CommandFactory cf = new BasicCommandFactory(null,this);
13
14     public Shell(boolean suppress)
15     {
16         output = new BufferedWriter(new OutputStreamWriter(System.out));
17         input = new BufferedReader(new InputStreamReader(System.in));
18         if (suppress)
19         {
20             prompt = "";
21         }
22     }
23
24     public void switchCommandSource(CommandFactory cf)
25     {
26         this.cf = cf;
27     }
28
29     public void operate() throws IOException
30     {
31         output.write(prompt);
32         output.flush();
33         String commandLine = input.readLine();
34         int space = commandLine.indexOf(" ");
35         String com;
36         String args = "";
37         if (space == -1)
38         {
39             com = commandLine;
40         }
41         else
42         {
43             com = commandLine.substring(0,space);
44             args = commandLine.substring(space+1);
45         }
46         com = com.toUpperCase();
47         try
48         {
49             Command c = cf.generateCommand(com,args,cf);
50             String[] out = c.execute();
51             for (int i = 0; i < out.length; i ++)
52             {
53                 output.write(out[i] + "\n");
54             }
55             output.flush();
56         }
57         catch (Exception e)
58         {
59             output.write("Unexpected Exception");
60             e.printStackTrace();
61             output.flush();
62         }
63
```

```
    }

    public static void main(String[] args) throws Exception
    {
        boolean suppress = false;
        if (args.length >  0)
        {
            suppress = true;
        }
        Shell s = new Shell(suppress);
        while (true)
        {
            s.operate();
        }
    }
}
```

```
public class SystemAlreadyExistsException extends Exception
{
    String name;
    public SystemAlreadyExistsException(String name)
    {
        this.name = name;
    }

    public String toString()
    {
        return "Filename " + name + " is already being used";
    }
}
```

```
public class UnknownCommand extends Exception
{
    protected String command;

    public UnknownCommand(String command)
    {
        this.command = command;
    }

    public String toString()
    {
        String str = "";
        str += "The Command: " + command+ " is not recognised";
        return str;
    }
}
```

Appendix D

Alternative Experiment Design

This appendix describes an alternative experiment design that was considered in place of the experiment described in chapter 6. This experiment was piloted with 7 subjects and invaluable insight was gained into the issues surrounding experimental design in relation to the goals stated in section 6.4.1.

D.1 Design

There are two basic task types: Enhancing (Enh) and Documenting (Doc) the code. Subjects will be asked to perform either the same type of task twice, or one and then the other. This gives 4 basic groups, Enh—Doc, Doc—Enh, Enh—Enh and Doc—Doc. Due to some subjects being asked to repeat the same type of task there must be two sets of Enhancement tasks and two different parts of the code to document. As a result, the code will be split into two sections, A & B, and when asked to perform the Document task the subjects will be told to document either section A or B. The Enhancement tasks will labelled C & D. This results in there being a total of 12 different groups: Doc(A)—Doc(B), Doc(B)—Doc(A), Enh(C)—Enh(D), Enh(D)—Enh(C), Doc(A)—Enh(C), Doc(A)—Enh(D), Doc(B)—Enh(C), Doc(B)—Enh(D), Enh(C)—Doc(A), Enh(C)—Doc(B), Enh(D)—Doc(A) and Enh(D)—Doc(B). Subjects will be assigned to these groups by stratified random sampling. The different possible 'paths' the subjects can take through the experiment are shown in figure D.1.

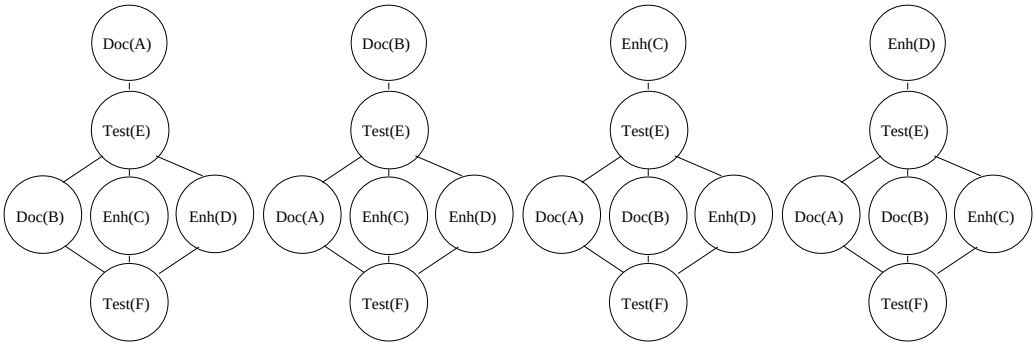

Figure D.1: The Twelve Different Experiment Groups

D.2 Measure of Level-of-Understanding

There are two measures of level-of-understanding. Firstly, there are the tests. The subjects' accuracy and time spent answering the questions will be used as a measure of level-of-understanding. The other measure is the performance of the second task as linked to the first task undertaken by a subject. The first task undertaken is a 'calibration' task, and obviously no measurement of level-of-understanding can be taken from how a subject performs in the task by itself. The key measurement is taken by finding how much better their work on the second task is as compared to others who performed the same second task. In this way it should be able to be determined if performing Documenting or Enhancing has an effect on the subjects' ability to perform the next task. There is also a 'check-sum' within the experiment, in that by comparing the quality of work the subjects produce with their test scores, this should be able to ascertain how accurate the tests are as a measure of subjects' level-of-understanding about the program.

To measure the quality of documentation, each comment will be rated in two areas: description of the function's purpose and the description of its arguments, with fractional marks being available for partial descriptions. Each part will have a difficulty rating of 1 to 3. The marks received for the descriptions will be multiplied by the difficulty for a score. Adding up the scores will give a documentation rating for the subject. A similar system will be used for Enhancement. Each Enhancement task will be given a difficulty rating of 1 to 3. For each task they complete the subject is given a score calculated by taking the difficulty of the task and dividing it by the time they took. Partially completed tasks will be given partial marks. Marks will be deducted if the enhancement breaks current functionality.

D.3 Conclusion

This experiment design is large and unwieldly, requiring a very large number of subjects to be able to gather accurate results, furthermore those results are based on subjective evaluation of the subjects' work. However, of great value was the pilot that was run, the insights of which, including material construction, subject selection and task selection, informed the final experiment design reported in chapter 6.

Appendix E

Interview Questions

Maintenance Programmer Questions

Section A Questionnaire

I) Programmer Background

Name?[I]

_____

Age? [I]

18–25 26–35 36–45 45–55 55+

Gender? [I]

Male Female

Job Title? [I]

_____

How long have you spent professionally programming? [I]

<1 year 13 years 38 years 8+ years

Which languages are you proficient in? [I]

Java C C++ Cobol SQL VB Perl C#

Others:_____

How many other systems/projects have you worked on? [I]

None 1 2–3 3–5 5–8 8+

How long have you spent maintaining this system? [I]

<1 year 1–3 years 3–8 years 8+ years

Did you also help develop the system? [I]

Yes No

II) System Background

How old is the software system that you work on? [I]

<1 year 1–3 years 3–8 years 8–15 years 15+

III) How are they assigned work?

Is there a particular part of the system that you consider yourself an expert on? [I]

Yes No

IV) Tools

What tools do you currently use in maintaining the system? [I]

Section B – Interview Questions

I) Programmer Background

Look over what they've written in the questionnaire to make sure they are happy with it. [I]

II) System Background

Source languages used in system? [I]

State of documentation? [D]

How many programmers maintain the system? [I]

What is the structure of the team maintaining the system? [D]

What is your position in the team? [I]

Could you draw a sketch of the system you work on? [D]

III) How are they assigned work?

Can you *briefly* explain the lifecycle of a maintenance request? [D]

What are the typical type of errors that you are asked to fix when performing maintenance? [I]

What is the time frame for you fulfilling a maintenance request? What is the shortest, longest and average time for performing maintenance? [D]

Do you only perform maintenance on that area of the system that you consider yourself expert in? [I] (*Only asked if they consider themselves an expert in part of the system – obviously)*

What happens when a critical flaw is found in the system? [D]

Is the lifecycle you described earlier always stuck too or is it shortcircuited to speed up fixes to the system? [D]

IV) What information do they use?

Apart from the source code what information do you use when maintaining the system (e.g. documentation, other people, you team, change logs, code traces etc). [D]

What do you need to know about the other systems that interact with the one you maintain? [D]

Do you have a personal (Database) of information about the system? [I]

What notes do you keep about the system? [D]

Are there project guru's that you can consult? [I] If so what kind of questions do you ask them? [D]

V) How do they get the information?

How do you go about finding out information you say you use in the previous section? Do you have to schedule meetings, are their certain tools you use etc? [Long D]

VI) How do they use the information?

How often are the answers to your problems simply "in the code" as opposed to answers worked out from other sources. [D]

When you read the source code while trying to carry out a maintenance request are there always certain things you look for every time. Are there key areas of the code that you can start from? [D]

VII) What info are they missing?

Brief discussion of tools they say they use from questionnaire. [D]

Do you use any self developed tools/scripts to help you maintain the system (what info do they give you)? [I]

What is your biggest problem in maintaining the system? [D]

When maintaining the system what information do you constantly find you require? [D]

Is there any information you need that you don't look up because it's too time consuming/difficult? [D]

What kind of tools/info do you find are missing when you maintain the system? [D]

Bibliography

[1] F. Abbattista, F. Lanubile, G. Mastelloni, and G. Visaggio. An experiment on the effect of design recording on impact analysis. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 253–259, Washington, DC, USA, 1994. IEEE Computer Society.

[2] A. Abran and H. Nguyenkim. Analysis of maintenance work categories through measurement. In *ICSM'91 Internaional Conferance on Software Maintenance*, 1991.

[3] E. Arisholm, H. Gallis, T. Dybå, and D. Sjøberg. Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Enginering*, 33(02):1000–9999, 2007.

[4] R. Backus. Applying instructional systems development to software maintenance education. In *ICSM'88: International Conferance on Softwre Maintenance*, 1988.

[5] C. Berg and S. Smith. Assessing students' abilities to construct and interpret line graphs: disparities between multiple-choice and free-response instruments. *Science Education*, 1994.

[6] L. Berlin. Beyond program understanding: A look at programming expertise in industry. In *Empirical Studies of Programmers, Fifth Workshop*, pages 6–25, 1993.

[7] B. Boehm. *The High Cost of Software*. Addison-Wesley, 1975.

[8] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *J. Syst. Softw.*, 80(4):571–583, 2007.

[9] L. C. Briand, V. R. Basili, Y.-M. Kim, and D. R. Squier. A change analysis process to characterize software maintenance projects. In *Proceedings of theInternational Conference on Software Maintenance*, pages 38–49. IEEE Computer Society Press, 1994.

[10] A. Brown, A. Christie, and S. Dart. An examination of software maintenance practices in a u.s. government organization. *Journal of Software Maintenance: Research and Practice*, 7(4), 1995.

[11] E. Burch and H.-J. Kungs. Modeling software maintenance requests: a case study. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, pages 40–47, Washington, DC, USA, 1997. IEEE Computer Society.

[12] J. M. Carroll. Making use: a design representation. *Commun. ACM*, 37(12):28–35, 1994.

[13] M. Cartwright and M. Shepperd. An empirical view of inheritance. Technical report, University of Bournemouth, 1998. TR98-02.

[14] J. Carver, J. V. Voorhis, and V. Basili. Understanding the impact of assumptions on experimental validity. In *International Symposium on Empirical Software Engineering*, pages 251–260, 2004.

[15] F. A. Cioch, M. Palazzolo, and S. Lohrer. A documentation suite for maintenance programmers. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, pages 286–295, Washington, DC, USA, 1996. IEEE Computer Society.

[16] B. Curtis. By the way, did anyone study any real programmers? In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 256–262, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[17] J. Daly, A. Brooks, J. Miller, and M. Wood. Evaluating the effect of inheritance on the maintainability of object-oriented software. In *Empirical Studeis of Programmers: Sixth Workshop*, pages 39–57, 1996.

[18] S. Dekleva. Delphi study of software maintenance problems. In *ICSM'92*, 1992.

[19] S. M. Dekleva. The influence of the information system development approach on maintenance. *Management Information Systems*, Sep 1992.

[20] I. S. Deligiannis, M. Shepperd, S. Webster, and M. Roumeliotis. A review of experimental investigations into object-oriented technology,. *Empirical Software Engineering*, 7(3):192–231, Sep 2002.

[21] A. Dunsmore and M. Roper. A comparative evaluation of program comprehension measures. Technical Report EFoCS-35-.

[22] S. G. Eick. *Software Visualization*, chapter 21 - Maintenance of Large Systems. The MIT Press, 1998.

[23] S. D. Fay and D. G. Holmes. Help! i have to update an undocumented program. In *CSM'85: Conferance of Software Maintenance*, pages 194–202, 1985.

[24] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*.

[25] M. A. Francel and S. Rugaber. The relationship of slicing and debugging to program understanding. In *Seventh International Workshop on Program Comprehension*, pages 106–113, 2000.

[26] R. Glass, I. Vessey, and V. Ramesh. Research in software engineering: an analysis of the literature. *Information and System Technology*, 44, 2002.

[27] R. L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley, 2003.

[28] R. L. Gorsuch. *Factor Analysis*. 1983.

[29] T. Graves and A. Mockus. Inferring change effort from configuration management data, 1998.

[30] R. Harrison, S. Counsell, and R. Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *J. Syst. Softw.*, 52(2-3):173–179, 2000.

[31] D. Heimbold and D. Luckham. Debugging ada tasking programs. *IEEE Software*, 2, March 1985.

[32] J. Hughes and S. Parkes. Impact of verbalization upon students software design and evaluations. In *EASE'04: Emprical Assesment of Software Engineering*, 2004.

[33] A. Hutton and R. Welland. An experiment measuring the effects of maintenance tasks on program knowledge. In *Empirical Assesment of Software Engineering*, 2007.

[34] M. Jorgensen and D. I. K. Sjøberg. Impact of experience on maintenance skills. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(2), 2002.

[35] M. Kajko-Mattsson, A. G. Glassbrook, and M. Nordin. Evaluating the predelivery phase of iso/iec fdis 14764 in the swedish context. In *ICSM'01: Proceedings of the 2001 International Conference on Software Maintenance*, 2001.

[36] J. D. Kalbfleisch and R. L. Prentice. *The Statistical Analysis of Failure Time Data*. Wiley, 2002.

[37] B. Kitchenham. Procedures for undertaking systematic reviews. Technical Report TR/SE-0401, Department of Computer Science, Keele University, 2004.

[38] B. Kitchenham, S. L. Pfleeger, and N. Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12):929–944, 1995.

[39] D. Kleinbaum and M. Klein. *Survival Analysis: A Self Learning Text*. 2005.

[40] J. Koskinen. Software maintenance costs. http://www.cs.jyu.fi/ koskinen/smcosts.htm.

[41] P. J. Layzell and L. A. Macaulay. An investigation into software maintenance-perception and practices. *Journal of Software Maintenance: Research and Practice*, 6(3):105–120, 1994.

[42] M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9), 1980.

[43] M. M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. London: Academic Press, 1985.

[44] B. Lientz and E. Swanson. *Software Maintenance Management*. 1980.

[45] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, 1978.

[46] S. Microsystems. Java pet store. http://java.sun.com/developer/releases/petstore/.

[47] A. Mockus and L. Votta. Identifying reasons for software changes using historic databases. In *ICSM'00: Proceedings. International Conference on Software Maintenance*, pages 120–130, 2000.

[48] M. Munro. Software maintenance, reuse and reverse engineering. In *Reuse maintenance and Reverse Engineering of software: Current practice and new directions*, 1989.

[49] G. Myers. A controlled experiment in program testing and code walkthroughs/inspections. *Communication of the ACM*, 1978.

[50] J. Neyman and E. Pearson. The testing of statistical hypotheses in relation to probabilities a priori, 1933.

[51] J. T. Nosek and P. Palvia. Software maintenance management: changes in the last decade. *Journal of Software Maintenance*, 2(3):157–174, 1990.

[52] P. Oman and C. R. Cook. Typographic style is more than cosmetic. *Communications of the ACM*, 33(5):506–520, May 1990.

[53] T. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. 1996.

[54] T. M. Pigoski and C. S. Looney. Software maintenance training: Transition experiences. In *ICSM '93: Proceedings of the Conference on Software Maintenance*, pages 314–318, Washington, DC, USA, 1993. IEEE Computer Society.

[55] C. J. Poole, T. Murphy, J. W. Huisman, and A. Higgins. Extreme maintenance. *icsm*, 00:301, 2001.

[56] V. Rajlich, J. Doran, and R. Gudla. Layered explanations of software: a methodology for program comprehension. In *IEEE Third Workshop on Program Comprehension*, pages 46–52, 1994.

[57] R. Razali, C. F. Snook, M. R. Poppleton, P. W. Garratt, and R. J. Walters. Experimental comparison of the comprehensibility of a uml-based formal specification versus a textual one. In *EASE'07: Emprical Assesment of Software Engineering*, 2007.

[58] C. Robinson. *Experiment, Design and Statistics in Psychology*. Penguin, 1994.

[59] H. Sackman, W. J. Erikson, and E. E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Commun. ACM*, 11(1):3–11, 1968.

[60] S. Schach, B. Jin, L. Yu, G. Heller, and J. Offutt. Determining the distribution of maintenance categories: Survey versus measurement. *Empirical Software Engineering*, 8(4):351–365, December 2003.

[61] S. R. Schach. *Software Engineering – 2nd Edition*. 1993.

[62] S. B. Sheppard, E. Kruesi, and J. W. Bailey. An empirical evaluation of software documentation formats. In *Proceedings of the 1982 conference on Human factors in computing systems*, pages 121–124, New York, NY, USA, 1982. ACM Press.

[63] S. E. Sim and R. C. Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *International Conference on Software Engineering*, pages 361–370, 1998.

[64] J. Singer. Practices of software maintenance. In *International Conferance on Software Maintenance*, 1998.

[65] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 21. IBM Press, 1997.

[66] D. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovi, N.-K. Liborg, and A. C. Rekdal. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Enginering*, 31(09):733–753, 2005.

[67] D. I. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanovic, E. Koren, and M. Vokác. Conducting realistic experiments in software engineering. In not found, editor, *ISESE'2002 (First International Symposium on Empirical Software Engineering)*, pages 17–26. IEEE Computer Society, 2002.

[68] I. Sommerville. *Software Engineering*. Addison Wesley, 7th edition, 2004.

[69] J. Spolsky. Hitting the high notes. http://www.joelonsoftware.com/articles/HighNotes.html.

[70] M. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins, and H. A. Müller. On designing an experiment to evaluate a reverse engineering tool. In *Working Conference on Reverse Engineering*, pages 31–40, 1996.

[71] M.-A. Storey, K. Wong, and H. A. Muller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2–3):183–207, 2000.

[72] E. Swanson. The dimension of maintenance. In *Proceedings of 2nd International Conference on Software Engineering*, pages 492–497, Oct 1976.

[73] E. B. Swanson, 2007. In e-mail conversation.

[74] M. J. Taylor, E. P. Maoynihan, and A. Laws. Training for software maintenance. *Journal of Software Maintenance*, 10(6):381–393, 1998.

[75] T. Thelin. Team-based fault content estimation in the software inspection process. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 263–272, Washington, DC, USA, 2004. IEEE Computer Society.

[76] W. F. Tichy, P. Lukowicz, L. Prechelt, and E. A. Heinz. Experimental evaluation in computer science: a quantitative study. *J. Syst. Softw.*, 28(1):9–18, 1995.

[77] G. Udny Yule. On the Theory of Correlation for any Number of Variables, Treated by a New System of Notation. *Royal Society of London Proceedings Series A*, 79:182–193, May 1907.

[78] H. van Vliet. *Softawre Engineering Principles and Practices*. John Wiley and Sons, 2nd edition, 2000.

[79] M. Vokác, W. Tichy, D. I. K. Sjøberg, E. Arisholm, and M. Aldrin. A controlled experiment comparing the maintainability of programs designed with and without design patterns – a replication in a real programming environment. *Empirical Software Engineering*, 9(3):149 – 195, Sep 2004.

[80] A. von Mayrhauser and A. Vans. Program understanding – a survey, 1994. Technical Report CS-94-120, Colorado State University, August 1994.

[81] A. von Mayrhauser and A. M. Vans. Program understanding behavior during debugging of large scale software. In *ESP '97: Papers presented at the seventh workshop on Empirical studies of programmers*, pages 157–179, New York, NY, USA, 1997. ACM Press.

[82] A. von Mayrhauser and A. M. Vans. Program understanding behavior during enhancement of large-scale software. *Journal of Software Maintenance: Research and Practice*, 9, 1997.

[83] A. von Mayrhauser and A. M. Vans. Program understanding during software adaptation tasks. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 316, Washington, DC, USA, 1998. IEEE Computer Society.

[84] E. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):929–944, 1988.

[85] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Kluwer Academic Publishers, 2000.

[86] M. V. Zelkowitz and D. R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, 1998.

[87] *International Conference on Software Maintenance*, 1985-2001.

[88] *European Conference on Software Maintenance*, 1997-2001.

[89] *Journal of Software Maintenance and Evolution: Research and Practice*, 1989-2001.