An Experiment Measuring the Effects of Maintenance Tasks on Program Knowledge

Alistair Hutton and Ray Welland
University of Glasgow , Department of Computing Science
17 Lilybank Gardens
Glasgow, Lanarkshire, G12 8QQ, UK
(huttona)(ray)@dcs.gla.ac.uk

Abstract

Objective: To ascertain whether programmers gain more knowledge about an unfamiliar program by enhancing the code or documenting the code. The context of this work was investigating whether maintenance programmers faced with an unfamiliar system should start by actively working on the system or spend time passively exploring the system before attempting to make changes.

Method: We designed a laboratory experiment where subjects initially either enhanced or documented a program and then we measured how they performed when carrying out a further task on the given code. Our hypothesis was that programmers would gain more knowledge performing one of the two tasks. The experiment was repeated three times with different groups of students, all at the same stage of their education.

Results: There was no significant difference between the performance of the two groups who had performed different initial tasks. However, there was a strong correlation between performance in the measured task and the students' programming ability, as measured by a previous academic assessment. As not all subjects completed the measured task within the given time, we needed to use Kaplan-Meier survival curves and the Cox Proportional Hazard Model to analyse our data. Detailed inspection of the code produced during the experiment revealed some interesting qualitative results.

Conclusions: We were unable to show a significant difference between the value of enhancing or documenting code as a way of gaining knowledge about unfamiliar programs. In the context of software maintenance this means that there is no advantage in spending unproductive time documenting code to gain knowledge.

Keywords: Experiment, Software Maintenance, Program Comprehension

1. INTRODUCTION

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, re-assignment or retirement) and new recruits joining either as direct replacements for departed members or because the workload of a group has increased. A useful term to describe these new additions to a team is *software immigrant* [7]. 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, 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 [8, 5] and if the outgoing team member is not available, the immigrant is on their own in terms of learning about the specifics of the sub-system that they are responsible for. Documentation is unreliable [6] 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.

Therefore, we thought it was worthwhile to construct an experiment to examine if performing different tasks with an unfamiliar piece of code caused any difference in the level-of-understanding gained by the programmer.

1.1. Related Work

There are a number of experiments that have a similar justification to our experiment but a wholly different goal. We are not directly interested in how programmers go about understanding code, whereas the work of Mayrhauser and Vans [9, 11, 12] which has a similar setup is an attempt to validate their mental model of program comprehension by examining what mental actions programmers take while performing maintenance tasks. We are approaching this area from the other end: trying to examine which tasks give a greater *level-of-understanding* rather than concerning ourselves with *how* the knowledge is gained. Of course, if one of the tasks we select does seem to give a greater level-of-understanding then we can examine what it is about the task that gives the beneficial effect and try and tie that to the various models of program comprehension that exist. That is a longer-term goal, and we are directly concerned with providing immediate benefit for maintenance teams and programmers.

2. EXPERIMENT DESIGN

2.1. Basic Overview

Subjects were split into two groups. Both groups undertook a given task for 1 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 was taken as their measure of level-of-understanding.

2.2. Hypothesis

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

Hypothesis: There is a significant difference in the level-of-understanding between the Document and Enhancement groups.

2.3. Subjects

Subjects were 4th year Computing Science students at the University of Glasgow. This meant each subject had 3 years' programming education with at least 1 year of programming experience in Java and a grade in a Java based programming module. Subjects were offered 20 pounds to participate in the experiment.

2.4. 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 task upon. They were then given as much time as necessary to read over a written specification of the system and any questions the subjects had about the system were answered. Subjects were then given one hour to perform the relevant Initial task (either Enhancing or Documenting) on the system. The subjects were then given a 10 minute break in which snacks and drinks were given to them and 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 is an enhancement task. The subjects were given at most 1 hour to finish the second task, and the length of time taken to successfully complete the Measured task is used as the metric of their understanding of the code.

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.

2.5. Materials

We used a single piece of code, details of which are provided below. A written specification of the system along with a basic class diagram of the code was provided. An example piece of documentation was required to show the Document group what level of detail was being looked for. We needed a specification on an enhancement task for the Initial Enhancement group and an enhancement specification to act as the Measured task. Finally a written description of the three tasks was also required.

3. JUSTIFICATION OF EXPERIMENT COMPONENTS

The subjects were provided with a basic class diagram. It showed only class inheritance hierarchies and interface implementation. We 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 to learning about an object oriented system. Given the prevalence of class diagram generating tools we thought it was only reasonable to provide subjects with a basic class diagram. Subjects were not allowed to use an IDE for the tasks. We decided on this as we felt that an IDE gives too many advantages to a proficient user over a subject not using an IDE. Due to the standard syllabus the subjects had undertaken, we knew that they would be familiar and competent with a number of standard text editors and program compilation from the command line. Whilst this restriction reduces the realism of the experiment we thought it was more important to keep the subjects more closely homogeneous when it came to productivity tools.

We feel that there is very little that can be usefully said about how a programmer works in general by studying how they work on a 15 line program. This means that the code we produced had to be of a 'reasonable' size, where reasonable means a sufficient size to require more than a superficial reading of the code to be understood. This is a very subjective decision and is dependent on the time constraints of the experiment and the complexity of the code. From piloting a previous version of the experiment we settled on a program of approximately 1,500 lines of code created within the domain and complexity bounds detailed below.

The program domain must be sufficiently simple or well known so that there is no advantage for a subject who is an expert in the field. The necessity of this is clear: a complex domain gives a domain expert a head start in understanding a program that is modelling some problem in the domain. We decided that the program would be 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, we feel there is no problem about domain complexity confounding the results of the experiment.

This experiment is not meant to test programming ability, so the program used was relatively 'straightforward', by which we mean that there is nothing algorithmically complex about the code, and that there are no tricks or deliberate traps placed in the code. This is not to say that the code is either simplistic or exceedingly well written: it was deliberately written with inconsistent naming conventions, different bracketing styles and inefficient algorithms, to try and simulate the disjointed nature of 'real-world' code. The code must be sufficiently modularised to allow the subjects to generate mental abstractions of the code as they read through and understand it. This is because the construction and use of mental abstractions of code has been identified as one of the most important 'tools' that a programmer has in working with code [10], and if the code did

not provide the necessary structures to allow subjects to relatively quickly abstract we would be removing one of the primary tools of their knowledge acquisition.

4. THREATS TO VALIDITY

4.1. Two Enhancements

The greatest threat to validity in the experiment relates to the two Enhancement tasks. If they are too similar then we are training the Enhancement group 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. Initially the experiment was piloted with a different Measured task to the one that was used in the final experiment. However, none of the 3 subjects who performed the pilot were able to complete the original Measured task. There were four possibilities: 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, we came to the conclusion 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. We considered two options for correcting this problem: additional explanation of the Measured task or a new Measured task that had lower domain complexity. We were operating under a time constraint, in that we only had access to the subjects for 2 weeks before their coursework started to mount up and they became unavailable to do the experiment. We felt on balance that the current Measured task was going to be too domain complex even after providing additional explanation and we could not take the risk of losing even more time. As a result we felt that producing a new Measured task was the safest approach to take. The Measured task we designed had a considerably lower domain complexity but it is still spread across the whole system thus requiring a wide range of knowledge about the system to complete. This new task does have a greater similarity in characteristics to the Initial task than we would have liked. 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, we do not feel that the two Enhancement tasks significantly bias the results.

4.2. Bug

There was a missing feature in the experiment code. Some subjects noticed and brought our attention to 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, we decided to take no action about it. We neither fixed the code nor drew special attention 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, we do not think any subject or group of subjects was disadvantaged by the bug. For the further replications of the experiment we decided to not fix the bug. Our justification is that we wanted totally unaltered materials to avoid introducing an extra (however small) threat to validity into the experiment.

4.3. Time Constraint and Statistics

As described in the experiment description, subjects were given one hour to complete the Measured task. This introduces a cut-off point which affects the statistical tests that can be used to analyse the completion times. For example, the standard t-test relies on having times for all the subjects. Our expert advice lead us to using survival analysis, and specifically, Kaplan-Meier survival curves with the log-rank test to examine the results [4]. The main fields of research in which survival analysis comes from are medicine and engineering, where they are used to measure time-to-event data, where the event might be patient death, component failure or remission. The concept is generic and can be directly applied to our experiment and can measure

time-to-completion of the Measured task. The key feature of survival analysis is that it allows computations on what is termed *right censored data*: in our case, subjects for whom there is no completion time. We just know they have not finished in the given time and thus we cannot say anything more than their completion time is greater than 60 minutes. As we cannot fit them into, say, Student's t-distribution, we cannot use those types of statistical tests, but simply eliminating the non-completing students from the statistical tests introduces a fairly obvious selection bias to the results.

The Kaplan-Meier survival curves combined with the log-rank test is a conceptually simple approach. It compares the distance of the actual observed event times from the theoretical event times if all the measurements come from the same population, i.e. the null hypothesis is true. The greater the distance the greater the chance that the null hypothesis is not true. The end result is a standard p-value, which we will be considering at $p < 0.05$. We discuss the statistical results in section 5.2. We provide the plotted survival curves in figure 1. Normally as a study progresses 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. In our specific case all censoring happens at 60 minutes, the end of our graphs, making our survival curves more straight forward to read.

We also have another angle of analysis. By examining the modifications the subjects made to the program in detail, as well as debriefing the subjects after the experiment, we can identify any trends, common features or interesting anomalies. We discuss these elements in section 5.3.

5. DISCUSSION OF RESULTS

5.1. Reasons for Repeating the Experiment

We ran the experiment 3 times, in 3 different years. The reason for this was that the first time we ran the experiment we received no significant result for our hypothesis but as can be seen from table 1 only 6 of the 10 Enhancers from the first run of the experiment (cohort) finished the Measured task (and all in 30 minutes or less) while 9 out of 10 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 an 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, we decided to run the experiment again with a further 18 subjects (2 Enhancers pulled out at the last minute). As can be seen from table 3, the 2nd cohort Enhancers seem to be of a different character to the 1st cohort Enhancers with a 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. We reviewed how we had run the experiment to try and identify any factors that may have sped up or slowed down the two Enhancement groups. We did not find any variation in how we ran the experiment so we also looked at external factors: the way that the programming courses that subjects had been taught might have changed and a more detailed examination of the subjects academic results was undertaken just in case the grades were 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 done the previous year and the more detailed grade analysis gave no further insight. As a result we ran the experiment a third time to try and find if there was a trend toward bi-modality of results in the Enhancers. We ran this third iteration of the experiment with 10 Enhancers and 5 Documenters.

5.2. Quantitative Results

In comparing the Enhancers with the Documenters using a survival curve we find $p=0.771$ as seen in table 2. This clearly leaves no significant difference between the Enhancers and Documenters

and as such we cannot reject the Null Hypothesis for the experiment. As can be seen from figure 1-a Enhancers hold an initial advantage over the Documenters but the late proliferation of times drags the group back. Testing for differences between cohorts reveals absolutely no significant differences ($p=0.996$) as is clear from figure 1-b. However, the survival analysis based on the subjects' programming grade, shown in figure 1-c is a significant indicator of completion time with $p=0.015$. To analyse the subjects' self-rating we split them into two groups, '6 and below' and '7 and above'. This gives $p=0.017$, although this hides some strange fluctuations, for instance the 8s are better than the 6s, but not the 5s. This does raise some doubt as to the ability of subjects to rate themselves. As a result we also used the self-rating as a univariate indicator in a Cox Proportional Hazard Model to determine if it was a significant indicator of the completion time and we received a p-value of 0.021 which is a significant result. However, we also analysed all variables together in a multivariate Cox Proportional Hazard Model, which produced the result that only grade was a significant indicator of completion time, with self-rating having a p-value of 0.158. This suggests that grade is a far more important indicator of completion time than self-rating and once you balance for grade the self-rating does not indicate much. 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 4 & 5 shows 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, which allows us to say that we have not biased the groups by ability.

Cohort	Initial Task	Time	Grade	Rating	Cohort	Initial Task	Time	Grade	Rating
1	Document	31	C	6	1	Enhance	12	A	8
1	Document	52	A	6	1	Enhance	24	C	5
1	Document	16	A	8	1	Enhance	Non-Comp	B	5
1	Document	58	B	7	1	Enhance	Non-Comp	C	6
1	Document	34	A	7	1	Enhance	19	A	9
1	Document	Non-Comp	C	7	1	Enhance	24	B	7
1	Document	27	A	8	1	Enhance	Non-Comp	A	9
1	Document	47	B	7	1	Enhance	9	A	7
1	Document	36	B	6	1	Enhance	30	B	6
1	Document	29	A	9	1	Enhance	Non-Comp	None	8
2	Document	25	A	8	2	Enhance	Non-Comp	C	7
2	Document	Non-Comp	C	2	2	Enhance	24	A	7
2	Document	33	A	7	2	Enhance	39	A	6
2	Document	Non-Comp	B	5	2	Enhance	21	B	8
2	Document	17	A	5	2	Enhance	Non-Comp	C	4
2	Document	19	A	8	2	Enhance	59	C	7
2	Document	56	B	6	2	Enhance	19	A	7
2	Document	17	A	7	2	Enhance	51	A	6
2	Document	43	A	7	3	Enhance	39	B	7
2	Document	34	A	7	3	Enhance	28	A	7
3	Document	60	C	7	3	Enhance	26	A	6
3	Document	Non-Comp	A	6	3	Enhance	30	C	7
3	Document	52	E	4	3	Enhance	23	A	5
3	Document	43	B	7	3	Enhance	10	A	7
3	Document	Non-Comp	A	6	3	Enhance	Non-Comp	C	6
					3	Enhance	52	B	7
					3	Enhance	41	A	6
					3	Enhance	16	A	7

TABLE 1: Subjects' Details

Variable	Chi Square	DF	p-value
Initial Task	0.131778	1	0.717
Cohort	0.008772	2	0.996
Grade	10.0706	2	0.007
Self-Rating	5.65624	1	0.017

TABLE 2: Log-Rank Tests

5.3. Qualitative Discussion

To allow us to discuss the results more fully, a brief description of the code is required. As described above it is a command line interface to a number of alterable sports ranking systems. The program 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

Group	Mean Time To Completion	Median Time To Completion
1st Cohort Enhance	23.8	24
2nd Cohort Enhance	41.375	39
3rd Cohort Enhance	31.7	28
1st Cohort Document	38.8	34
2nd Cohort Document	35.6	33
3rd Cohort Document	55	60

TABLE 3: Mean and Median Time to Completion

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 4: 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 5: Number of Subjects by Self-Rating

FIGURE 1: Survival Curves

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 3 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.

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: 2 of these could be described as being close to finishing, 1 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 10 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 change to get a working program would be fairly minimal *if only* they had a clear 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 & 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 10 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, 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 no 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 understanding and it could be said that they were caught on a technicality, 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 100% sure about how they worked so did not use them. 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.

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 the Measured task, the subjects who made a significant effort all adopted different approaches. There is a fairly rough link between the amount of work done in the Initial Enhancement task and the subjects' performance in the Measured task, 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. Similarly for the Document task there was a rough link between the number of classes commented and quality of the comments in relation to how well they performed on the Measured task. There were 8 classes that we defined as being 'key' to the operation of the system: the majority of the Documenters commented these classes. Four of the 5 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.

Over the years in which the experiment was run we noticed a shift in the subjects' use of the computing environment. Although the subjects all still had at least 1 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.

6. FURTHER WORK

As with all experiments, external replication is important and we would hope that other researchers would want to try and repeat our experiment. As identified in section 4 the composition of the program and tasks are a confounding factor in this experiment. We think it would be an important step to replicate this experiment with different materials to see if the same result holds true. We would certainly want a greater difference between the Initial Enhancement and Measured tasks. Specifically we think that one of the tasks should be additive, in that new functionality should be added to the system, while the other task is changeative, in that current functionality is altered. In our experiment both tasks were additive in nature which means in practice adding a command to the system, which might be thought of as being too similar. However we would argue that adding a command is one of the the most knowledge intensive tasks that can be performed and only a subject with a high level-of-understanding of the system could perform the task.

As programming is a specialised task the quality of the subjects is very important. It has been argued from the very beginning [2] that testing undergraduates is no substitute for using professional programmers. It is generally much harder to get professional programmers to perform controlled experiments, although the Simula Research Lab has extensive experience in obtaining professional subjects for their experiments [1] and if the resources necessary are available it seems the best approach to take. Our experiment is also fairly short for the topic that we are investigating. If the subjects were given more time for the initial tasks on a larger program it could be that one or other of them will prove itself superior.

Finally, these two Initial tasks should be compared with subjects who have access to a mentor. Such an experiment would be tricky to design as one person's mentoring is another person's boring lecturing. Once there is some agreement on what mentoring should consist of then we could compare a short amount of mentoring time, say half an hour, against the full hour of Enhancing or Documenting.

We would not change the fundamentals of the experiment. While there are alternative ways of measuring a subject's level-of-understanding of a piece of code, they all have negative attributes which we think make them inferior to using a programming task. Dunsmore and Roper [3] lay out the various ways of measuring a subject's level-of-understanding. We feel that it is very hard to compose a set of questions that reveal a subjects level-of-understanding about the code, especially when you want to introduce time to answer as well as accuracy of answer into your results. Metrics that rely on recall without access to the code are confounded by the subject's quality of memory. Getting the subject to explain the code to an expert who judges

their level-of-understanding is confounded by how articulate the subject is. The major benefit of using a programming task to measure level-of-understanding despite its indirect nature is that programming ability is what we expect programmers to have, and in the case of undergraduates, can be carefully controlled.

7. CONCLUSION

We have designed and run a solidly constructed experiment 3 times and have applied robust, appropriate statistical tests to our data to derive our results. In the short term there is no difference between enhancing and documenting code as a means of acquiring knowledge about the program. We can say that the Glasgow Advanced Programming course assessment system is a very accurate way of reflecting Java programming ability, as measured by our experiment.

In the context of software maintenance, in the absence of a mentor, software immigrants should start working with the code and be given maintenance tasks to perform. From our experiment it seems they learn just as much about the code by working with it as they do from hanging back and taking a more general view. As a result, by being thrown into the deep end, as it were, they will start producing useful work at least as quickly as if they had taken time out to learn the system. As both the Enhancers and Documenters failed in a similar fashion in the experiment there is no apparent benefit to having adopted either approach when it comes to poorer programmers.

Acknowledgements: We would like to thank Professor John McColl of the University of Glasgow for his expert advice on the statistical tests used and the 53 students who made the experiment possible.

REFERENCES

- [1] Benestad, H.C. and Arisholm, E. and Sjøberg, D. (2005) How to Recruit Professionals as Subjects in Software Engineering Experiments. Information Systems Research in Scandinavia, August 6-9, Kristiansand Norway
- [2] Curtis, B.(1986) By the way, did anyone study any real programmers? 1st Workshop on Empirical Studies of Programmers
- [3] Dunsmore, A. and Roper, M. (2000) A Comparative Evaluation of Program Comprehension Measures. University of Strathclyde - TR
- [4] Kalbfleisch, J.D. and Prentice, R.L. (2002) The Statistical Analysis of Failure Time Data. Wiley
- [5] Pigoski, T. and Looney, C.S. (1993) Software Maintenance Training: Transition Experiences. 9th International Conference on Software Maintenance
- [6] Singer, J. (1998) Practices of Software Maintenance. 14th International Conference on Software Maintenance
- [7] Sim, S.E. and Holt, R.C. (1998) The Ramp-Up Problem in Software Projects: A Case Study of How Software Immigrants Naturalize. 14th International Conference on Software Engineering
- [8] Taylor, M. and Maoynihn, E. and Laws, A. (1998) Training for Software Maintenance. Journal of Software Maintenance, Vol: 10 Num: 6
- [9] von Mayrhauser, A. and Vans, A. (1994) Comprehension processes during large scale maintenance. 16th International Conference on Software Engineering
- [10] von Mayrhauser, A. and Vans, A.(1994) Program Understanding – A Survey. Colorado State University - TR
- [11] von Mayrhauser, A. and Vans A.(1997) Program understanding behavior during debugging of large scale software. 7th Workshop on Empirical Studies of Programmers
- [12] von Mayrhauser, A. and Vans, A. (1997) Program understanding behaviour during enhancement of large-scale software. Journal of Software Maintenance: Research and Practice, Vol: 9 Num: 5