An Experiment on the Effect of Design Recording on Impact Analysis

F. Abbattista, F. Lanubile, G. Mastelloni, and G. Visaggio

Dipartimento di Informatica
University of Bari, Italy

Abstract

An experimental study is presented in which participants perform impact analysis on alternate forms of design record information. The primary objective of the research is to assess the maintainer performance with respect to various approaches of design recording. Among the approaches there is the model dependency descriptor which includes decision capturing and explicit traceability links between software objects and decisions. Results indicate that design recording approaches slightly differ in work completeness and time to finish but the model dependency descriptor leads to an impact analysis which is the most accurate. These results suggest that design records have the potential to be effective for software maintenance but training and process discipline is needed to make design recording worthwhile.

1: Introduction

The need to find better ways to collect information on development and maintenance activities is essential for the software evolution. The concept of design record is the answer to the information gap that maintainers suffer in doing their work. In this sense, we adopt the working definition of design recording, given in [1], where a design record is defined as a collection of information with the purpose to support activities following the development phase.

Everybody advocate to collect information about software products and software processes as they evolve. However, as in the past for the structured programming and structured design, little empirical research has been done to show that the availability of design records makes easier the maintenance work and produces more reliable results.

The objective of this paper is to investigate how differences in design recording influence maintenance performance, and more specifically if decision capture and traceability support result in measurable improvements in maintenance performance.

To isolate only those maintenance aspects which are important with respect to the evaluation goal of this study, we focused on the impact analysis as one phase of the maintenance process. Impact analysis is the activity of identifying what to modify to accomplish a change, or of identifying the potential consequences of a change [2]. In our software maintenance model, impact analysis starts when a programmer is given a change request and finish with identifying which modules change.

Next section briefly describes the approaches to design recording which have been compared. Section 3 presents the experiment and section 4 shows the results. Last section suggests some conclusions and indicates issues for future research.

2: Design recording approaches

The more traditional way to record design information is to use some standard for software development which requires the writing of documentation. For example IEEE standards on software requirements specification, software design description, and software test documentation cover the products of the phases of analysis, design, and testing. Two main problems arise with such kind of documentation. The former is the large volume of information which can make difficult the consultation for a maintainer. The latter is the lack of rationales behind the choices of the original developers.

An alternative approach may be a system which stores decisions and traceability links to navigate into the documentation and rationales. In [5], a design recording model, called *model dependency descriptor*, was presented to support software maintenance. The model dependency descriptor describes a software system as a web which integrates the different work products of software life cycle and their mutual interrelationships. Rationales are distinguished between design and implementation decisions. A decision is represented as a problem to be addressed, a set of alternatives, the adopted solution, and the justification for it. Decisions are linked to software objects involved in the transformation process. For example, a design decision links objects from specification requirements to objects in the design description, while an implementation decision connects objects from the design description to objects of the implementation. Decisions are also linked each other to represent relationships of cause, justification, generalization, and succession.

The third approach to design recording works as a control group because it represents the total lack of documentation. The only material which is available to maintainers is the source code. This is a very common situation in the real world of software maintenance where, even when the documentation is present, it is often out of date and then useless.

3: The experiment

We hypothesize that different design recording approaches significantly affect impact analysis. To verify this hypothesis we conducted a blocked subject-project study by comparing dependent variables across a set of subjects, a set of maintenance tasks, and a set of software systems. This was a controlled experiment but with tasks and software systems significant with respect to the real maintenance world.

3.1: Participants

The experiment involved 23 computer science students from a fourth year course in software engineering at the University of Bari. We did not use software programming professionals because we could not involve enough subjects for handling the high heterogeneity which characterize the programming behavior. Since using few subjects there is a risk of obscuring the experiment with subject differences [4], we preferred to use students as subjects of this experiment.

All students had gained experience with Structured Analysis/Structured Design and Pascal programming from a practical software engineering project in a previous university course. For the experiment, students were given a series of lectures on the model dependency descriptor.

3.2: Software systems

Three versions of an information system for a publishing company were chosen to be the targets for the maintenance tasks in the experiment. Each system was based on the same specification but with different design and code, produced by second year computer science students as an assessed part of their degree course. The systems were implemented in Pascal to run on a IBM-compatible PC.

Table 1 provides some metrics to describe the product characteristics. Because of the program size it was not possible for subjects to understand all the details. Thus, from a maintenance perspective, the systems can be considered representative of realistic conditions.

	Sys1	Sys2	Sys3
# design decision	36	18	21
# impl decision	20	22	21
# data files	15	13	16
# source code files	44	40	14
# proc/function	181	117	272
LOC	6636	6593	9684

Table1. Comparison of system characteristics

3.3: The maintenance tasks

Each subject received three requests of change, one for each software system, upon to perform impact analysis. The criteria which have been taken into account for selecting the requests were the following.

First, required changes should be realistic as representative of actual user demands. We satisfied this requirement choosing requests among a list suggested by students in a prior course on software testing. Second, the related impact analysis should be executed in a reasonable time. In our case we needed to complete a task in less than two hours. The pilot study was used to adjust the order of complexity of maintenance tasks. Third, the maintenance tasks should be representative of the wide range of possible changes. We choose to select one task for each type of maintenance: corrective, adaptive, and perfective. Finally, the maintenance tasks should be identical for all the three systems and for each kind of design record information. This constraint was respected because of the common requirements and hardware/software platform.

The corrective maintenance task required the programmer to change the systems to check the acquisition of telephone numbers. The systems accept all kind of characters and no validation is done before storing. Task 1 required to accept only numeric digits and "/"s for separating the country code, the area code and the local number.

The adaptive maintenance task required to modify the time for acquiring the current date from the operating system. The systems read the current date in an initialization routine at the start of the program. Task 2 required to read the current date at the start of each transaction since the program could run without stops for all the weekdays.

The perfective maintenance task required the programmer to modify the organization of files. The systems use only sequential files as allowed in Pascal. Task 3 requires to use Btrieve libraries for changing to indexed-sequential organization so that system performance could be improved.

3.4: Experimental design

A partially confounded 3×3×3 factorial design with repeated measures was used in the experiment. All the factor combinations were used and each subject was exposed to each of the factor levels. Figure 1 shows a schematic representation of the experiment plan. The symbol *Gi* represents the i-th group of subjects. Seven groups were made up of three subjects and two groups of two subjects. Subjects in each group were observed under a combination of the three factors. The first factor was the task to perform, the second factor was the software system upon to perform impact analysis, and the third factor was the design record information which was available: (1) only source code, (2) standard documentation, and (3) model dependency descriptor. The numbers in parentheses represent the respective levels for the factor combination. For example (112) means that the subjects were observed under a combination of the first task, the first software system, and the second level of the design recording factor (standard documentation). Observations were arranged as three Graeco-Latin squares so that the sequence of treatments was balanced to control for the learning effect. Repetitions allowed to neutralize the effects of individual differences on performance.

The primary purpose of this plan was to obtain complete within-subject information on all main effects. However, only partial information can be obtained on all the interaction effects because they are partially confounded with differences between groups. A detailed description of this type of design can be found in [8, p.566].

Programmer performance was measured by three variables:

- the completeness of the estimated impact
- the accuracy of the estimated impact
- the time taken to analyze the change impact

Completeness and accuracy were measured by comparing the actual impact set and the estimated impact set. The former, the *actual impact set* (AIS), is the set of objects which would be actually modified by the execution of the change request. AIS is not necessarily unique, since it depends by the hypothesis of solution to implement the change. The latter, the *estimated impact set* (EIS), is the set of objects which programmers estimate be affected by the change.

Completeness is measured as the percentage of objects in AIS which are recognized by programmers and put in EIS. Completeness is zero when the AIS and EIS are disjoint, while it is equal to one when AIS is contained in EIS. If AIS is empty then the completeness is undefined.

On the other hand, accuracy is measured as the percentage of objects in EIS which belong to AIS. Accuracy is zero when the two sets are disjoint, while is equal to one if EIS is contained in AIS. Accuracy is undefined when EIS is empty. The desired situation is when the two sets are equal. In this case both completeness and accuracy are equal to one.

G1	(112)	(231)	(323)
G2	(133)	(222)	(311)
G3	(121)	(213)	(332)
G4	(132)	(221)	(313)
G5	(123)	(212)	(331)
G6	(111)	(233)	(322)
G7	(122)	(211)	(333)
G8	(113)	(232)	(321)
G9	(131)	(223)	(312)

Figure 1. Experiment plan

3.5: Experimental procedure

As a preliminary exercise all subjects were presented with the same Pascal program with the three levels of design recording. The purpose was to decrease learning effects prior the real experimentation tasks, to ensure that change requests were reasonable complex and experiment procedures contained no ambiguities.

Students performed impact analysis on three change requests upon the same software system, obviously different from the experiment objects. The three maintenance tasks were administered sequentially on increasing levels of design recording. Time assigned for each task was 45 minutes. A short break was allowed between tasks.

The pilot study revealed that the 45 minute time limit should be extended. Some subjects had problems in understanding the maintenance tasks and others in using the model dependency descriptor to make impact analysis. As a result a supporting lecture was given on design recording in software maintenance and the correct impact analyses were shown to students. We also answered many questions about the experimental procedure.

Following the pilot study, subjects were presented for three times in different days with one modification request, one software system, and one design recording approach. They were allowed 90 minutes to finish the impact analysis, during which they filled in a data collection form containing object types be affected by the change.

Level 1 of the documentation factor included only the source listing of the program. Level 2 included also the software requirements specification and the software design description with relational schema of data, structure charts, module descriptions, and the layouts of the user-interface. Level 3, in addition to this documentation, included the design and implementation decisions, and a graphical drawing of the model dependency descriptor which showed external traceability. Internal traceability was not explicitly represented in the graph but could be extracted using the documentation of the second level.

The experiment employed manually-based techniques rather than automated techniques to avoid the effects of unequal training to supporting technologies.

Once the impact analysis task was completed, the subjects were asked to give a score to the task and program comprehension. Programmers were also encouraged to write comments about the experience.

4: Analysis of results

Before proceeding to a statistical analysis, collected data were analyzed looking for doubtful data points. We used the criteria, already applied in [7], of discarding data when subjects showed a misunderstanding of experiment instructions or an unawareness of basic programming concepts. The consequence was that 13 out of a total of 69 impact analyses were discarded as being too incomplete or inaccurate to produce reliable results.

When presenting the following performance statistics, we use some acronyms for the levels of the design recording factor. *Code* is the first factor level consisting in managing only a source code listing, *Std Doc* is the second factor level consisting of standard documentation, and *MDD* is the third factor level representing the model dependency descriptor.

In table 2, the completeness appears to increase with added levels of design records. However, the differences between design recording approaches are small. Using the first level of the design record factor as a base of comparison, completeness improved of 7% with standard documentation and 14% with the model dependency descriptor. Looking at the tasks, only the second task took a clear advantage of having design record information besides the source code listing, while the performances in the third task were degraded. The reason is that in the model dependency descriptor there were decisions which directed maintainers in doing their work, while there were no decisions directly related to the issues of the third task.

Table 3 stresses the difference of accuracy obtained with the model dependency descriptor with respect to others. Accuracy with standard documentation worsened of 4% with respect to source code, but it improved of 39% with the model dependency descriptor. In this case too, the second task was the most improved by the use of decisions and traceability links.

	Code	Std Doc	MDD
Task1	0.59	0.63	0.69
Task2	0.19	0.48	0.51
Task3	0.94	0.75	0.80
Overall	0.58	0.62	0.66

Table 2. Completeness across tasks and design recording approach

	Code	Std Doc	MDD
Task1	0.71	0.69	0.81
Task2	0.30	0.37	0.83
Task3	0.46	0.34	0.41
Overall	0.49	0.47	0.68

Table 3. Accuracy across tasks and design recording approach

Analysis of variance was used to test the significance of the differences. The results for the accuracy factor are presented in table 4. All the computed F statistics fall below the critical values both for alpha = 0.05 and alpha = 0.25. This means that there are no significant differences with multiple comparisons. Since looking at the means of the design recording approaches we noted a 39% improvement with the model dependency descriptor but the overall F test was non-significant, we made a posteriori comparison between every pair of means for the levels of the design recording factor. Table 5 shows the results of the pairwise tests. At the significance level 0.25, the hypotheses *Code = MDD* and *Std Doc = MDD* are rejected while *Code = Std Doc* is accepted.

The poor performance of standard documentation can be explained as a 'confusion effect' that subjects suffered because of large volume of information to deal with. On the contrary, impact analysis was better with the model dependency descriptor since it helped maintainers to navigate into design record information.

SOURCE OF VARIATION	SS	df	MS	F
Between Subjects	7.65	13.75		
Groups	0.73	8.00		
(Task × DRA)'	0.17	2.00	0.08	
(Task × System)'	0.17	2.00	0.09	
(DRA × System)'	0.10	2.00	0.05	
(Task × DRA × System)'	0.28	2.00	0.14	
Subjects within groups	7.20	5.73	1.26	
Within Subjects	8.28	29.45		
Task	0.76	2.00	0.38	0.43*
DRA	0.41	2.00	0.20	0.23*
System	0.88	2.00	0.44	0.50*
(Task × DRA)''	0.23	2.00	0.12	0.13*
(Task × System)''	0.46	2.00	0.23	0.26*
(DRA × System)''	0.15	2.00	0.07	0.09*
(Task × DRA × System)''	0.59	6.00	0.10	0.11*
Error (within)	10.07	11.45	0.88	

* $F_{.95}(2,11) = 3.98$; $F_{.75}(2,11) = 1.58$
** $F_{.95}(6,11) = 3.09$; $F_{.75}(6,11) = 1.55$
Legend: DRA = design recording approach

Table 4. Analysis of variance: accuracy variable

Task × DRA		DRA × System	
H	F	H	F
Code = Std Doc	0.01	Code = Std Doc	0.01
Code = MDD	3.10	Code = MDD	4.78
Std Doc = MDD	2.85	Std Doc = MDD	4.41

$F_{.95}(1,2) = 18.5$; $F_{.75}(1,2) = 2.57$

Table 5. Accuracy: Individual comparisons of design recording approaches

Table 6 shows time performance measured in minutes. A slight additional time, 13% more than only source code, was needed to perform impact analysis with the model dependency descriptor. This can be explained with the decisions to read and the less familiar model to record design. On the other hand, time decreased of 6% with standard documentation with respect to source code. This result seems to confirm the confusion effect which induced maintainers to a superficial impact analysis.

Subjective scores on system and task comprehension are generally optimistic and exhibit a uniform distribution over the design recording factor and the other independent variables. Although their interpretation is doubtful, we want to stress the analogy with results in [6] where structural differences between programs were not discernible to programmers.

	Code	Std Doc	MDD
Task1	58	47	49
Task2	71	63	82
Task3	57	63	78
Overall	62	58	70

Table 6. Time across tasks and design recording approach

5: Conclusions and future research

The purpose of this investigation was to compare maintenance performance in the presence of different approaches to design recording. We used a design description document, as a form of traditional documentation, and we tested the model dependency descriptor, which makes use of design rationale and traceability links. As a control group we measured performance with only source code available.

Results from the experiment indicate that using the model dependency descriptor significantly improves the accuracy of impact analysis. Completeness improves too but not with the same difference, while time to finish slightly increases.

Programmers were not able to manage with effectiveness the standard documentation, due to the large volume of available information. On the contrary, using the model dependency descriptor, which lets to recognize the developer's intentions and explicitly shows links between software objects and decisions, the maintenance work resulted more precise. This supports our confidence in the usefulness of this design recording approach for the maintenance of large applications.

We expect better results in the future, by changing the maintenance process model which our subjects implicitly follow. As most maintainers in the real world, our students have become skilled in taking the source code and make the necessary changes first to the code and later, but not always, to the accompanying documentation. For this approach, called quick-fix model in [3], the added design record information is not a help but a weight to suffer. Intensive training and a careful process discipline are needed to change the old habits.

Readers should also note that although our subjects received lectures on design recording, they had experience only with constructing own documentation and not with consulting other people's documents.

We want also to stress that our students did not use any supporting technology to compensate the added volume of information. The lack of appropriate tools could have penalized the approaches which use increasing levels of design record information.

We had to discard several measures due to misunderstandings or inadequate skills. Although the multifactor design of the experiment neutralizes individual differences in their ability, more training and more severity in subject recruitment is needed.

Before the results can be used outside we have to revalidate them in different environments and, at the same time, we have to correct experiment deficiencies. Our commitment in the future research is for replicating the experiment using both students and experienced programmers, university classrooms and industrial projects, more training, a thorough control of the maintenance process, and tools to manage design record information.

Acknowledgements

This work was supported by "Progetto Finalizzato Sistemi Informatici and Calcolo Parallelo" of Italian National Research Council (CNR) under grant no.91.00930PF69.

We would like to thank students at the University of Bari for their hard work which made this experiment possible. Thanks also to anonymous referees for their helpful comments.

References

[1] R.S.Arnold, M.Slovin, and N.Wilde, "Do design records really benefit software maintenance?",

Proceedings of Conference on Software Maintenance 93, Montreal, Canada, IEEE Computer Society Press, 1993, pp.234-243.

[2] R.S.Arnold, S.A.Bohner, "Impact analysis - towards a framework for comparison", *Proceedings of Conference on Software Maintenance 93*, Montreal, Canada, IEEE Computer Society Press, 1993, pp.292-301.

[3] V.R.Basili, "Viewing maintenance as reuse-oriented software development", *IEEE Software*, January 1990, pp.19-25.

[4] R.Brooks, "Studying programmer behavior experimentally: the problems of proper methodology, Communications of the ACM, vol.23, n.4, 1980, pp.207-213.

[5] A.Cimitile, F.Lanubile, G.Visaggio, "Traceability based on design decisions", *Proceedings of Conference on Software Maintenance 92*, Orlando, Florida, IEEE Computer Society Press, 1992, pp.309-317.

[6] V.R.Gibson, and J.A.Senn, "System structure and software maintenance performance", *Communications of the ACM*, vol.32, no.3, March 1989, pp.347-358.

[7] M.Shepperd, and D.Ince, "Design metrics and software maintainability: an experimental investigation", *Software Maintenance: Research and Practice*, vol.3, 1991, pp.215-232.

[8] B.J.Winer, *Statistical principles in experimental design*, Mc-Graw Hill Book Company, Inc., 1962.