MAINTENANCE OF OBJECT ORIENTED SYSTEMS

AN EMPIRICAL ANALYSIS OF THE PERFORMANCE AND STRATEGIES OF PROGRAMMERS NEW TO OBJECT-ORIENTED TECHNIQUES

J. van Hillegersberg

K. Kumar

Department of information scienc. Rotterdam School of Management. Erasmus University

G1-40 PO.Box 1738. 3000 DR Rotterdam

The Netherlands. Email jhillegersberg@fac.tbk.eur.nl

AND

R..J. Welke

Computer Information Systems dept..Georgia State University

P.O. Box 4015m. Atlanta Georgia. USA

EXTENDED RESERACH ABSTRACT

NOVEMBER 2. 1994

ABSTRACT

The structured paradigm for software development did not solve the software maintenance problem. Currently object-orientation (OO) is viewed as a main opportunity to improve maintenance productivity. Although some promising results haven been reported. other studies conclude that understanding and maintenance of OO systems can be difficult. Development and maintenance of OO systems is often performed by developers which were originally trained in structured languages and techniques and are relatively new to OO. This paper describes the results of a controlled experiment designed to evaluate the maintenance strategies and productivity of such developers. The experiment concludes that programmers with experience in structured development but with low experience in OO development have trouble understanding and maintaining an OO system.

1. Introduction

"For every dollar you spend on software development you will spend two dollars on software maintenance (Boehm, 1987) ". This quote clearly shows the importance of software maintenance. It also reveals that the currently dominant structured paradigm for software development failed to reduce the huge effort required to maintain sotfware. The last decade, as systems became more complex, software maintenance started exceeding all other data processing division's efforts. Research shows that about 70% of the software budget is spent on maintanance of existing systems (Pfleeger, 1991).

As an alternative to the structured paradigm, the Object-Oriented (OO) paradigm emerged (Coad & Yourdon, 1991; Meyer, 1989; Booch, 1993).

OO promises to relief the maintenance-burden. A variety of claimed benefits of OO exists (Hillegersberg, 1993). The following claims directly relate to improved maintenance productivity:

- OO systems are easier to comprehend than Structured Systems (since encapsulation enforces system modularity & information-hiding)
- OO systems are easier to extend than Structured Systems (since inheritance supports extension)

Although a number of case and system development studies have been performed using OO there is hardly any experimental evidence for this assumption (Booch, 1993; Love, 1993).

To evaluate the impact of OO on software maintenance we designed a controlled experiment. This paper describes the preliminary results of this experiment.

2. Background and Related Work

2.1 Maintenance of Structured and Object-Oriented Systems

In the OO paradigm systems are decomposed based on their objects. Objects encapsulate both data and process. By organizing objects in classes, objects can inherit characteristics from their ancestors. The main technique in OO development is bottom-up system composition. The low-level objects are identified and organized into class-hierarchies.

An increase of maintenance produtivity should mainly be caused by the fact that OO systems are easier to comprehend than their structured counterparts. Comprehension of sofware by the programmer is critical, since is it a subtask of all other maintenance activities.

To comprehend a program, three actions can be taken: read about it (e.g. read documentation); read it (e.g. read the source code); or run it (e.g. watch execution, get trace data, watch dynamic storage etc). Although reading documentation and executing the program can be useful, the source code is often the primary and only accurate source of information. Today's programmers spend most of their time studying old source code before they can implement an enhancement (Corbi, 1989).

Structured source code typically consists of data structures and procedures which operate on the data to complete a certain process. If an enhancement requires a change in the data structure, this can effect many procedures. To add a new function, the maintenance programmer needs to have an understanding of the control flow of the program, that is, the calling hierarchy of the procedures.

OO source code consists of objects. Both the data and behavior of the object are specified. To understand the code a programmer will examine the objects. Since objects correspond to real world entities the programmer will have little trouble understanding their purpose. If a change is required the programmer will locate the appropriate object and change its behavior. As long as the object

interface remains unchanged. other parts of the system are not effected. Some enhancements will require the programmer to create a new object. In many cases this new object will resemble other objects allready present in the system. By using inheritance the programmer only has to specify the new behavior.

Maintenance of OO systems as described above is believed to significantly increase the productivity of developers. However. recent studies have also reported problems with maintenance of OO systems. Only few empirical- and field studies have been conducted to investigate differences in maintenance productivity on OO and structured systems. The following section will briefly review these studies. The remains of this paper describes an empirical study we conducted to investigate differences in maintenance on OO and structured systems.

2.2 Empirical studies comparing maintenance of Object-Oriented and structured systems

This section summarises empirical studies which have been conducted to compare OO and structured maintenance. Table 1 presents the main characteristics of these studies.

Henry and Humphrey (1990) let students add new features to object-oriented and procedural systems with identical functionality. They found the number of changes required for the OO code to be significantly lower. Also changes in the code were more localized for the OO program. Strong features of the experimental design are the large size of the programs used and the automatic data collection. However, the within-subject design requires all subjects to implement all tasks twice using C and Objective-C. Although subjects were told not to think about the OO and structured solution simultaneously, interaction effects can easily occur. The completion of a task is measured by running the adapted program using four sets of test data. This technique assures objective testing but fails in judging the quality of the modified code. As an example. a subject can extend an Objective-C program by writing some additional procedural code. The authors recognize the limitations of using students which are inexperienced in object-oriented

programming. However, they also argue that this bias gives even more support to the power of object-oriented programming.

Mancl and Havanas (1990) recorded maintenance activities on a telephone operation control system which consists partly of procedural C code and partly of C++ code. They investigated effects of all modification requests (adaptive, corrective, perfective) on structured and OO modules. The data shows a lower proportion of interface changes per modification request for the OO modules. This result seems to support enforced information hiding in the object-oriented parts of the system. Also the number of source code lines that had to be changed were registered. Especially for adaptive maintenance the object-oriented modules turned out to be more stable. The number of files changed per modification request was higher for the OO part of the system. This finding does not support the claim that the effects of modifications are more local in OO programs. The authors explain this by the intensive use of header files in the C++ language. Among strength of this research are the realistic setting and the large size of the system. Also the metrics only focus on the nature of the changes. No actual productivity data is recorded.

Wybolt (1990) reported on the object-oriented re-design and re-implementation of a commercial CASE-tool which was previously written in C. The main benefits were improved maintenance and reuse. Adding new methodology-support to the tool required 3.000 to 10.000 new C++ lines compared to 25,000 to 67.000 C lines for the original product. During maintenance of the re-engineered product navigating through the C++ code was difficult. The developers found inheritance to decrease encapsulation: "Inheritance does not necessarily isolate where functions can be found, nor does it localize their effect".

Wilde and Huitt (1992) collected project statistics and developers experiences on the software maintenance of three object-oriented systems at Bell Communications Research. They identify some concepts of object-orientation which may complicate high-level understanding of the system. First, the calling hierarchy of methods can be difficult to grasp. Dynamic binding makes a static analysis of the message chain impossible. Especially for beginning programmers

the absence of a real "main" method tends to be disconcerting. Second, finding where different functions are carried out can be difficult since functionality is dispered into different object classes. Wilde et al. suspect this problem to be even more serious than in conventional systems. Third, polymorphism can cause subtle errors. Several different implementations of the same method can lead to misinterpretation of the method by the maintainer.

Author(s)	Method	Developers	Language	Program Size	Dependent Variable	Results for OO
Henry, Humphrey (1990)	Laboratory study	senior level college students (24)	C / Objective-C	4000 loc	size/location of changes errors made perceived difficulty time spend	Requires less changes Changes are more local Perceived more difficult
Manci, Havanas (1990)	Case study	professionals	C / C++	> 100.000 loc	interface changed size/location of changes	Less changes of interface Changes are smaller Changes are not significantly more local
Wilde & Huitt (1990)	Field study / Survey	professionals	3 systems C++ / Smalltalk	500-2000 methods		Dynamic binding and inheritance complicate system understanding Changes can be dispersed among objects
Wybolt (1992)	Case study	professionals	C / C++	> 100.000 loc	size of changes perceived difficulty	Changes are smaller Chages are easier "Visual" navigation through C++ code is more difficult Class-inheritance violates encapsulation

Table 1. Studies comparing OO and structured maintenance

2.3 SUMMARY

The studies conducted report several advantages and problems in maintainance of OO systems compared to structured systems. The results of these studies are in many cases inconsistent. Some studies provide evidence for claimed benefits of

OO. Other observations reports on possible problems caused by OO concepts such as polymorphism and inheritance.

In the laboratory experiment described in the next sections we investigate maintenance productivity on OO systems compared to structured systems. By taking a closer look at the behavior of maintenance programmers we have collected data about the problems programmers have in maintaining OO systems.

3. Experimental Design

3.1 Research Model

The hypothesis of this study is that building systems in an OO manner increases maintenance productivity compared to structured systems. Higher maintenance productivity means that the programmer can make enhancements to a system of the same quality in less time. The underlying assumption is that programmers have less difficulty in understanding an OO system. By having a better understanding of the system, they will be able to change code more efficiently. Also the OO concept of inheritance should enable the programmer to extend a program with less effort.

The central research question is: What is the effect of development paradigm (OO / structured) on software maintenance productivity ? To investigate this question a laboratory experiment was designed. The research model is shown in Figure 1. The model depicts the relationships among the variables, tasks and subjects of the study.

3.1.1 Independent Variables

The main independent variable is the development paradigm (OO or structured). The other independet variable is the complexity of the task. The complexity can be low (only making a minor change in one code line) or high (adding a whole new class or procedure to the system).

3.1.2 Dependent Variables

Depent variables are productivity and problem-solving behavior. Also perceived difficulty, motivation and confidence in every task were measured. Productivity is measured by dividing a task quality score by the time required to complete the task. The perceived difficulty, motivation and confidence were obtained by giving the participants a post-experiment questionaire. The problem solving behavior was recorded automatically by the development environment specially built for this experiment.

3.1.3 Control variables

There may be other extraneous factors that can influence the relationship between the independent and dependent variables. Therefore the development environment, total time, general education level, developers experience and documentation are held constant during the experiment.

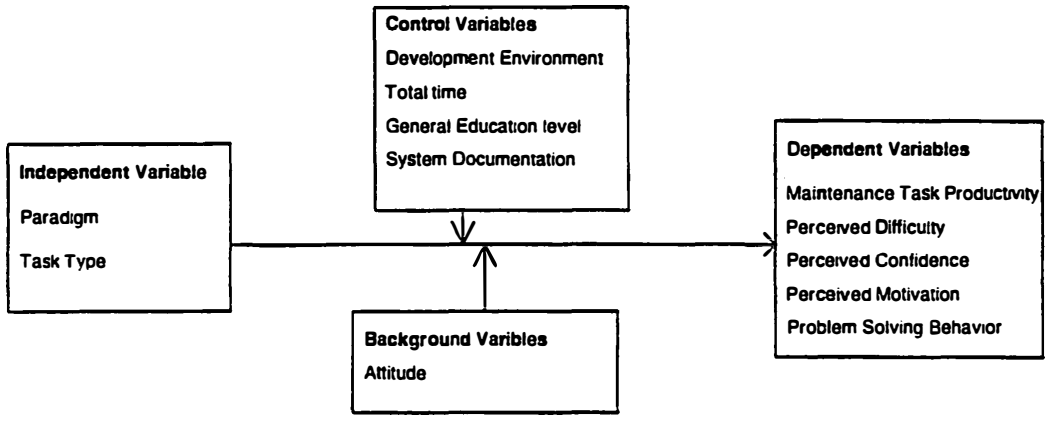

Figure 1. Research model

3.2 PARTICIPANTS

The subjects were 14 advanced graduate students enrolled in a software engineering course. Subjects had an average of 5-10 years experience in structured programming. Each subject knew about 6 structured analysis and design techniques and 2 structured programming languages. All subjects were novices in

OO design and implementation (6 monthes experience), knowing only one analysis and design technique (Coad/Yourdon) and one programming language (OO Pascal).

3.3 MATERIALS

Both systems to which the changes were made were coded from identical specifications. They were functionally identical so that when running, it was impossible to distinguish the programs.

System 1 was a payroll program which calculated salaries of different type of employees. The structured design and implementation of this system were adapted from a software engineering course book (Peters III, 1986). The OO design and implementation were developed by the authors. System 2 was a simulation program which calculated and animated the behavior of a industrial robot which picked-up randomly placed targets. The OO design and implementation of this system were adapted from a OO textbook (Lane, 1990). The structured design and implementation were developed by the authors. These programs were chosen for the experiment since they encompass a wide range of programming problems, administrative as well as process control. The size of both systems is listed in Table 2.. Developers also obtained a copy of the global design diagrams which showed the system decomposition in objects / procedures.

	Payroll		Robot Simulation	
	Structured	OO	Structured	OO
Lines of Code	160	340	360	490

Table 2. : Experimental matenals

3.4 PROCEDURE

A special development environment, called EXP, was built for conducting the experiment. Using EXP the subjects were able to do basic editing of the source

code. compile and run programs. see error messages and view the maintenance task they had to complete. EXP has a very simple graphical user-interface (see Fig 2..) which eliminates the effect of differences merely caused by the expertise of the developer with complex commercialy available development tools. EXP also registered how the subject navigated through the source code. what parts of the program were changed. what error messaged were received etc.

To maximize control. all subjects participated simultaneously in the computer-lab using a networked PC. All instructions and documentation (change requests. design diagrams etc.) were given by EXP. They were first presented a very simple "warming-up" task which consisted of making a simple change to a very small program. This task was used to let subjects gain familiarity with the EXP-environment.

Subjects could decide for themselves when they were satisfied with the modification. By pressing a "next task" button they were presented the next task.

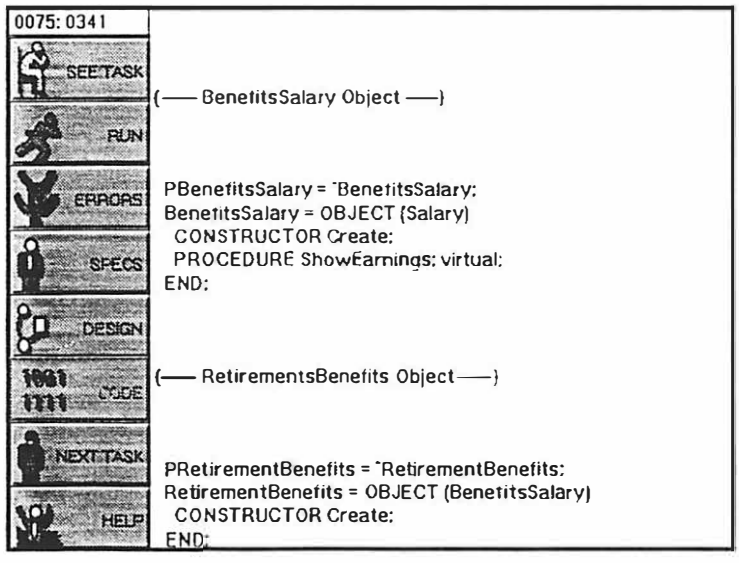

Fig 2.. The user screen of EXP.

All subjects performed enhancement maintenance on two systems. The two systems were designed and implemented using both OO and structured techniques which resulted in four programs.

All subjects performed seven tasks. The first task was a warming-up excercise to gain familiarity with the experimental environment. In this task all subjects made a small change to the same simple program. The remaining six tasks were organized as a "within subject" test. Subjects were randomly assigned to two treatment groups. Group A first completed three tasks on the OO implementation of the Payroll system. followed by three tasks on the structured implementation of the Simulation system. Group B first modified the structured version of system Payroll system. followed by the OO version of the Simulation System.

The maintenance tasks were enhancements to the systems varying from a simple enhancement to a full new feature. The task were presented to te developer as a user's request for change. This assured independence of the implementation language.

Subjects had a maximum of two hours to complete the seven tasks. Afterwards they were asked questions by EXP about their development and PC-experience. education and attitude towards OO methods. Also each maintenance task was again presented on the screen and the subject was asked to rate the difficulty, motivation and confidence he had when completing that task.

3.5 MEASUREMENT

Most of the data was automatically collected by EXP. Only the quality of the changes made by the programmer was judged independently by two experienced programmers. A maximum of two points were given for a correctly completed task. The scores were significantly correlated ($r = 0.76$, $p = 0.000$). Task productivity was defined as the average quality score divided by the time needed to complete the task.

4. Experimental Results

4.1 Completion times

Table 3. shows the average completion times for each task. Maintenance productivity is defined as:

$$productivity = \frac{score}{completion\_time}$$

Table 4. shows average productivity for each task. In task 1 all subjects performed the same task on the same program to gain familiarity with the EXP experimental environment. The completion times and productivity do not differ significantly which provides evidence for the assumption that the two treatment groups are equal.

Task 2 consisted in a small change in the user interface of the program. The main problem for the programmer here was to locate the part of the program which handled the user input. There were no significant differences in completion time and productivity for this task.

Task 3 and 4 both required the programmer to add a new type of employee to be handled by the program. For the OO implemetation this meant adding a new class making use of inheritance. For the structured implementation this meant adding a new record type and changing part of a function. For both tasks subjects were significantly slower and had a lower productivity on changing the OO system.

System	Task Nr	Mean Time [s] Structured	Mean Time [s] OO	p =
Warming-up	1	196.7	226.7	0.569
Payroll	2	1667.4	1087.6	0.325
Payroll	3	945.0	1643.7	0.012*
Payroll	4	725.9	1228.6	0.003*
Robot Simulation	5	598.7	1077.6	0.027*
Robot Simulation	6	1215.4	1694.4	0.372
Robot Simulation	7	525.0	331.8	0.334

Table 3.: Average task completion times and ANOVA significance (n=14. * indicates a significant result)

System	Task Nr	Mean productivty OO	Mean productivity Structured	p =
Warming-up	1	213.3	253.5	0.561
Payroll	2	61.1	30.3	0.257
Payroll	3	17.2	43.8	0.011*
Payroll	4	31.1	58.1	0.034*
Robot Simulation	5	40.0	68.3	0.023*
Robot Simulation	6	7.1	19.7	0.216
Robot Simulation	7	3.4	30.6	0.037*

Table 4.: Average maintenance productivity and ANOVA significance (n=14. * indicates a significant result)

Task 5 encompassed increasing the size of the target in the robot animation. This change required the programmer to locate the part where the target was drawn. Again the OO group was significantly slower and less productive.

In task 6 the programmer was asked to make a change in the algorithm which controlled the robot movement. This change could be made adding an extra function or method. No significant difference in productivity or completion times were recorded.

Task 7 asked the programmer to add a second target to the simulation and than let change the algorithm in such a way to let the robot pick-up the closest target first. This could be done by changing 2 methods / functions. A significant lower productivity for the OO group was found.

4.2 PROCESS ANALYSIS

The previous section showed that the OO group performs significantly lower on four out of six maintenance tasks. We expected, based on the literature, that subjects would be productive with OO after extensive training for about 6 month. Apparently shifting to OO is not that straightforward. To invest the problems the subjects had understanding and changing the OO programs we investigated the protocol data further. Our findings are summarized in this section. To limit the number of results we will concentrate on task 3 (Payroll) and 5 (Robot simulation) which both showed significant differences. Also in these two tasks the understanding of the program was vital to make the extension to the system.

5. Discussion

The assumption that OO concepts are very easy to learn and use is not true for the studied population. These programmers, which had extensive training in structured programming, and limited experience in OO development had more difficulties understanding and maintaining the OO systems than the structured systems. Similar results were found in other studies on OO maintenance. Future research will be focused on analysis of protocol data to learn more about why programmers have difficulties being productive with OO. Also other groups of subjects with more experience in OO development will be used in the experiment to find out more about the learning-curve associated with Oo development.

6. References

Boehm, B.W. (1987). Industrial software metrics top 10 list. IEEE Software, 4 (5) September, 84-85.

Booch, G. (1993). Object-oriented analysis and design with applications. (2nd ed.) Redwood city, California: The Benjamin/ Cummings Pub. company.

Coad, P. & Yourdon, E. (1991). Object-Oriented Analysis. (2nd ed.) Englewood Cliffs, NJ: Prentice Hall.

Corbi, T.A. (1989). Program understanding: Challenge for the 1990s. IBM Systems Journal, 28 (2), 294-306.

Henry, S.M., Humphrey, M. & Lewis, J.A. (1990). Evaluation of the maintainability of object-oriented software. In Anonymous (Ed.). Proceedings of the conference on computer and communication systems (pp. 404-409). Hong Kong:

Hillegersberg, J.v. (1993). Object-Oriented versus Structured software development: A controlled experiment. Management Report Series, 153, 1-32.Lane, A. (1990). Object-Oriented turbo Pascal. Redwood City, CA: M&T Pub..

Love, T. (1993). Object Lessons: Lessons learned in Object-Oriented development projects. New York, NY: SIGS Books, Inc.

Mancl, D. & Havanas, W. (1990). A study of the impact of c++ on software maintenance. In Anonymous (Ed.). Proc. Conf. software maintenance (pp. 63-69). Los Alamitos, Calif.: IEEE CS Press.

Meyer, B. (1989). From structured programming to object-oriented design: the road to Eiffel. Structured Programming, 1, 19-39.

Peters III, J.F. (1986). Problem solving with PASCAL, programming methods, algorithms, and data structures. New York, NY: CBS College Pub..

Pfleeger, S.L. (1991). Software engineering : The production of quality software. (2nd ed.) New York: MacMillan Pub. Comp..

Wilde, N. & Huitt, R. (1992). Maintenance support for object-oriented programs. IEEE Transactions On Software Engineering, 18 (12), 1038-1044.Wybolt, N. (1990). Experiences with C++ and Object-Oriented software development. ACM Sigsoft Software Engineering Notes, 15 (2), 31-39.