Information and Software Technology 51 (2009) 293–305

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

The effect of task order on the maintainability of object-oriented software

Alf Inge Wang [a,*], Erik Arisholm [b,c,1]

[a] Department of Computer and Information Science, Norwegian University of Science and Technology, Sem Sælandsvei 7-9, N-7491 Trondheim, Norway
[b] Simula Research Laboratory, PO Box 134, 1325 Lysaker, Norway
[c] Department of Informatics, University of Oslo, PO Box 1080, Blindern, N-0316 Oslo, Norway

ARTICLE INFO

Article history:
Received 6 October 2007
Received in revised form 5 March 2008
Accepted 11 March 2008
Available online 1 April 2008

Keywords:
Object-oriented design
Object-oriented programming
Maintainability
Maintenance planning
Software maintenance
Schedule and organizational issues

ABSTRACT

This paper presents results from a quasi-experiment that investigates how the sequence in which maintenance tasks are performed affects the time required to perform them and the functional correctness of the changes made. Specifically, the study compares how time required and correctness are affected by (1) starting with the easiest change task and progressively performing the more difficult tasks (Easy-First), versus (2) starting with the most difficult change task and progressively performing the easier tasks (Hard-First). In both cases, the experimental tasks were performed on two alternative types of design of a Java system to assess whether the choice of the design strategy moderates the effects of task order on effort and correctness.

The results show that the time spent on making the changes is not affected significantly by the task order of the maintenance tasks, regardless of the type of design. However, the correctness of the maintainability tasks is significantly higher when the task order of the change tasks is Easy-First compared to Hard-First, again regardless of design. A possible explanation for the results is that a steeper learning curve (Hard-First) causes the programmer to create software that is less maintainable overall.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

The effort required to make changes correctly to a software system depends on many factors. These factors include characteristics of the software system itself (e.g., code, design and architecture, documentation of the system, testability), the development environment and tools, the software engineering process used, and human skills and experience. For example, an empirical study by Jørgensen and Sjøberg [1] showed that the frequency of major unexpected problems is lower when tasks are performed by maintainers with a medium-level of experience than when they are performed by inexperienced maintainers. However, using maintainers with even greater experience did not result in any further reduction of unexpected problems. Further, the results of one of our previous empirical studies [2] showed that the effect of the design approach to a system on the time spent on, and correctness of, changes made depends not only on the design but also on the experience of the maintainers. In the study presented herein, we investigated how the breakdown and sequential ordering of maintenance tasks affects the time required to carry out change tasks and the resulting quality of the system. If we could find any indications that the way in which change tasks are ordered affects the maintainability of the system, the result would represent a high return on investment for software companies, because little effort is required to rearrange the task order.

In some cases, the priority of maintenance tasks is constrained by client priorities: *must have, good to have*, and *time permitting features/fixes* [3] or organisational goals [4]. In other cases, there are fewer constraints on how to break down and arrange the maintenance tasks, in which case one can choose freely among alternative strategies to prioritize or sequence the tasks. Thus, we wanted to assess the effects of ordering maintenance tasks with respect to *difficulty level*. We provide empirical evidence regarding two alternative strategies pertaining to the sequence of performing the change tasks: *Easy-First*, where the maintainers start with the easiest change task and progressively perform the more difficult tasks and *Hard-First*, where the maintainers start with the most difficult change task and progressively perform the easier tasks. Our decision to perform controlled experiments stems from the many confounding and uncontrollable factors that could blur the results in an industrial case study [5]. It is usually impossible to control for factors such as ability and learning/fatigue effects, and select specific tasks to assign to individuals in a real project setting. As a result, the threats to internal validity are such that it is difficult to establish a causal relationship between independent (e.g., Hard-First vs Easy-First) and dependent variables (e.g., time, correctness). We report results from two controlled experiments (which, taken together, form one quasi-experiment [henceforth,

* Corresponding author. Tel.: +47 73 59 44 85; fax: +47 73 59 44 66.
E-mail addresses: alfw@idi.ntnu.no (A.I. Wang), erika@simula.no (E. Arisholm).
[1] Tel.: +47 67 82 82 00; fax: +47 67 82 82 01.

0950-5849/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2008.03.005

the experiment]) that investigate whether the sequence of change tasks affects the correctness and the effort spent. Hence, the experiment attempted to assess two competing hypotheses:

(1) By starting with the easy maintenance tasks first, the learning curve will not be very steep, thus enabling the maintainers to obtain a progressively better overview of the software system before having to perform more difficult tasks. In this way, the maintainer is less likely to devise suboptimal solutions when performing the difficult tasks. This is closely related what is defined as the bottom-up strategy regarding program comprehension, in which programmers look for recognisable small patterns in the code and gradually increase their knowledge of the system [6].

(2) By starting with the difficult maintenance tasks first, the learning curve will be steep, as the programmer must obtain a more complete overview of the system before being able to perform changes. However, due to the better overview, the maintainer might be less likely to devise suboptimal task solutions. This is related to the top-down strategy regarding program comprehension, in which the programmer forms hypotheses and refinements of hypotheses about the system that are confirmed or refuted by items of the code itself [7].

The difficulty level of maintenance tasks is a nontrivial concept, and hard to predict precisely. In our experimental context, the difficulty level of the maintenance tasks was determined a priori by the authors, by considering the number and complexity of classes that would be affected by each change and an estimate of the time required to perform the tasks. The actual results of the experiment suggest that this approach was sufficiently accurate for the purpose of ranking the difficulty level of the tasks, as discussed further in Section 5.3.1. Such an approach might be amendable to industrial contexts as well, for example by performing a relatively informal and course-grained change impact analysis as a basis for the ranking.

The learning curve of a system also depends on how the system is structured. Hence, we also included two different design styles in the experiment: (i) a centralized control style design, in which one class contains most of the functionality and extra utility classes are used; and (ii) a delegated control style design, in which the functionality and data were assigned to classes according to the principles for delegating class responsibility advocated in [8]. These two design styles represent two extremes within object-oriented design. According to object-oriented design experts, a delegated control style is easier to understand and change than a centralized control style [9,10].

The software system to be maintained was a coffee vending machine [8]. The change tasks were relatively small (from 15 min to 2 h per task). In a case study of 957 maintenance or change requests in a company, Hatton found that above 75% these maintenance tasks lasted from 1 up to 5 h [11]. Admittedly, the experimental systems and tasks are small compared with industrial software systems and maintenance tasks. This is representative of a very common limitation in controlled software engineering experiments, but the impact of such limitations depends on the research question being asked and the extent to which the results are supported by theories that explain the underlying and more general mechanisms [12]. In this study, generalization of the results to larger systems and tasks can mainly be claimed based on the support of existing theories within program comprehension research (see Section 2). The results reported herein are supported by several of these theories (see Section 4.3). The impact on external validity are further discussed in Section 5.4.2, were we argue that the observed effects of task order on small systems might be *conservative* estimates of the effect that would be observed on larger systems.

The subjects were 3rd–5th year software engineering students who had no prior knowledge of the system being maintained. Given these experimental conditions, the results reported herein are not necessarily valid for experienced maintainers or maintainers who already have obtained a detailed understanding of the system that is to be maintained. However, we still believe that the scope of the study is highly relevant in an industrial context. This is because in our experience, it is common to assign new and inexperienced programmers to maintenance tasks, and unless careful consideration is given to the nature of the tasks assigned, such programmers may affect adversely the maintainability of the system. In addition, it has become more common to outsource the maintenance of a system to consultants who have no, or very little, prior knowledge of the system [13–15].

The remainder of this paper is organised as follows. Section 2 describes the theoretical background for the study. Section 3 describes the design of the experiment and states the hypotheses tested. Section 4 presents the results. Section 5 discusses what we consider to be the most important threats to validity. Section 6 concludes.

2. Maintainability of object-oriented software

The ISO 9126 [16] analysis of software quality has six components: functionality, reliability, usability, efficiency, maintainability, and portability. The ISO 9126 model defines maintainability as a *set of attributes that bear on the effort needed to make specified modifications*. Furthermore, maintainability is broken down into four subcharacteristics: analysability, changeability, stability and testability. However, these subcharacteristics are problematic in that they have not been defined operationally. Our experiment investigated how the process by which a software system is comprehended to perform changes affects maintainability; hence, it is principally the subcharacteristic analysability that is examined. Analysability is related to the process of understanding a system before making a change (program comprehension).

In addition to analysability, the experiment is related to changeability. Arisholm [17] views changeability as a two-dimensional characteristic: it pertains to both the *effort* expended on implementing changes, and the resulting *quality* of the changes. These (effort and resulting quality of changes) are also the quality characteristics we measured in the empirical study presented in this paper. There are several papers that describe studies that focus on making changes to a system ([18,19], and [20]). However, most of these studies focus on the *results* of changes to the software system and not the *process* of changing it, so they are not particularly relevant to our work. There are also papers that study how the relevant parts of the source code to be changed are found [21,22]. Our experiment only consider the effect of changing the sequence of change tasks in terms of required effort and resulting quality and does not report on how the individual maintainer makes the specific changes in the code.

In the following subsections, we elaborate upon the notion of analysability as it relates to software maintenance, program comprehension and we describes some empirical studies.

2.1. Analysability of object-oriented software

We define *analysability* as the degree to which a system's characteristics can be understood by the developer (by reading requirement, design and implementation documentation, and source code) to the extent that he can perform change tasks successfully. To be able to maintain and change a system efficiently (i.e., in a short-time) and correctly (i.e., with intended functionality and a minimum of side-effects) the maintainer must understand the sys-

Download English Version:

https://daneshyari.com/en/article/551992

Download Persian Version:

https://daneshyari.com/article/551992

Daneshyari.com