To Appear in *Information and Software Technology*

Human and Program Factors Affecting the Maintenance of Programs with Deployed Design Patterns[*]

T.H. Ng
City University of Hong Kong
Kowloon Tong, Hong Kong
cssamng@cityu.edu.hk

Yuen Tak Yu
City University of Hong Kong
Kowloon Tong, Hong Kong
csytyu@cityu.edu.hk

S.C. Cheung
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
scc@cse.ust.hk

W.K. Chan[†]
City University of Hong Kong
Kowloon Tong, Hong Kong
wkchan@cs.cityu.edu.hk

Abstract

Context: Practitioners may use design patterns to organize program code. Various empirical studies have investigated the effects of pattern deployment and work experience on the effectiveness and efficiency of program maintenance. However, results from these studies are not all consistent. Moreover, these studies have not considered some interesting factors, such as a maintainer's prior exposure to the program under maintenance.

Objective: This paper aims at identifying what factors may contribute to the productivity of maintainers in the context of making correct software changes when they work on programs with deployed design patterns.

Method: We performed an empirical study involving 118 human subjects with three change tasks on a medium-sized program to explore the possible effects of a suite of six human and program factors on the productivity of maintainers, measured by the time taken to produce a correctly revised program in a course-based setting. The factors we studied include the deployment of design patterns and the presence of pattern-unaware solutions, as well as the maintainer's prior exposure to design patterns, the subject program and the programming language, and prior work experience.

Results: Among the factors under examination, we find that the deployment of design patterns, prior exposure to the program and the presence of pattern-unaware solutions are strongly correlated with the time taken to correctly complete maintenance tasks. We also report some interesting observations from the experiment.

Conclusion: A new factor, namely, the presence of pattern-unaware solutions, contributes to the efficient completion of maintenance tasks of programs with deployed design patterns. Moreover, we conclude from the study that neither prior exposure to design patterns nor prior exposure to the programming language is supported by sufficient evidences to be significant factors, whereas the subjects' exposure to the program under maintenance is notably more important.

Keywords

Design patterns; human factors; pattern-deployed software; program factors; software maintenance; software requirements; software testing; software validation

Research Highlights

1. Six human and programs factors for the effective use of design patterns are studied.
2. Maintainers may not develop pattern-aware solutions even if they exist.
3. The presence of pattern-unaware solutions affects the time to complete a correct change task.
4. A prioritized list of the factors is proposed for further study.

1. Introduction

Modern approaches to software development, such as Prince2, Unified Process, or Scrum, recommend an iterative process. For instance, after producing the initial version of a program, software developers may modify the code to add more pieces of

[*] This work was supported in part by a Strategic Research Grant (project number 7008039) from City University of Hong Kong, and research grants (CERG project numbers 111410, 123207, and 612108) from the Research Grants Council of the Hong Kong Special Administrative Region, China.

[†] Contact author.

functionality or fix bugs in succeeding iterations. Examples include additional support of a new language or a new currency. Maintainers should properly manage such changes to maintain the code quality and keep the software intact [6]. One way to maintain the code base is to modify it to meet the current change requirement on an "as-needed" basis. An alternative is to project future change requirements, such as the support of more languages, and modify the code base to fulfill the current change requirement as well as the projected ones. In other words, the code base can be modified to enhance variability [13] for future change requirements. For instance, in a simplest form, maintainers may replace a hard-coded constant by a variable and consolidate the definition of the variable to one location in the code base.

Developers have adopted *design patterns* [15][16] to maintain the flexibility of object-oriented programs with respect to certain recurrent design problems [27]. For example, *CPPUnit* [9] uses the *Composite* pattern, *Microsoft COM* [36] uses the *Factory Method* pattern, and *J2SE JavaBeans* uses the *Observer* pattern [22].

Many studies have investigated the effective use of design patterns for software maintenance. In an early study reported in 2001, Prechelt et al. [39] studied whether using design patterns would lead to faster maintenance. Four small-sized programs were studied. The human subjects included those with and without prior training on design patterns. The deployment[1] of the *Abstract Factory*, *Composite*, *Decorator*, *Observer* and *Visitor* patterns was investigated. For simplicity, we refer to a program in which at least one design pattern is deployed as a *pattern-deployed program*.

In a few cases studied by Prechelt et al. [39], design patterns were found to be slightly unbeneficial, but the authors viewed the added complexity as the price for the flexibility provided by these deployed design patterns. In general, Prechelt et al. [39] concluded that, unless there is a clear reason not to deploy design patterns in programs, it is probably wise to provide the flexibility by using design patterns because unexpected new requirements often appear. Later, Vokáč et al. [46] replicated the study and reported some different results in 2004. For example, while Prechelt et al. [39] discovered a negative scenario on deploying the *Observer* pattern and a neutral scenario on deploying the *Visitor* pattern, Vokáč et al. [46] found a contrary result for *Observer* and a negative scenario for *Visitor*. Vokáč et al [46] concluded that design patterns are not universally good or bad, but must be used in a way that matches the problem and the people. Ng et al. [34] compared the efforts of maintaining programs with and without deployed design patterns by subjects having different amounts of work experience. They found that even inexperienced subjects working on programs with deployed design patterns could spend significantly less maintenance effort than experienced subjects do when working on programs without deployed design patterns.

The observation of inconclusive results from previous empirical studies suggests that certain important factors have not been considered. Software development is a highly creative process. Therefore, both human and program factors should be studied, as they are often intertwined with one another [41]. Nonetheless, in theory, there are an unlimited number of possible factors. We therefore focus on the three factors that have been studied in previous work [34][39][46], namely, deployment of design patterns, prior exposure to design patterns, and prior work experience. In this paper, we further consider three other factors, namely, the presence of pattern-unaware solutions, prior exposure to the program, and prior exposure to the programming language. In a certain sense, these additional factors are less immediately relevant than the first three factors. However, it is unclear, on top of the first three factors, how effective maintainers can complete the change tasks successfully subject to the influences of the three other factors. Interestingly, our finding to be presented in this paper reveals that prior exposure to the program does contribute to the effective completion of maintenance tasks.

In this paper, we report an empirical study that investigates the effects of the above six factors on maintainers' performance. Our experiment involves 118 human subjects in a course-based context. Each subject was required to perform three change tasks on *JHotDraw*, a medium-sized open-source program [23]. Based on the empirical results, we further prioritize the significance of the factors under study in terms of the time spent by maintainers, and discuss them. Specifically, we conclude from the study that neither prior exposure to design patterns nor prior exposure to the programming language is supported by sufficient evidences to be significant factors, whereas other factors such as the subjects' exposure to the program under maintenance are notably more important.

A preliminary analysis of the experimental results has earlier been presented in [35]. It reports an empirical study that involves human subjects and identifies two factors out of six having significant effects on the efficiency of a developer to complete a task of the maintenance nature correctly. Furthermore, we observe that more experienced maintainers tend to be able to complete a change task correctly without spending an amount of time that deviates significantly from the norm, even though they do not necessarily complete the task faster than less experienced maintainers on average do.

Compared with the preliminary version, this paper also substantially elaborates on the details of experimental setup and procedures, and specifically explains more clearly the motivation of the six factors selected for study and their intuitive relationships. It further reports a more systematic and in-depth analysis of the experimental results: (1) a detailed analysis of the effects of each factor leading to some hitherto unnoticed interesting findings, (2) a qualitative analysis of cases giving a clearer intuitive picture of the interplay of the factors, and (3) an extended statistical analysis of the quantitative results,

[1] The term "deployed design pattern" has been extensively used in our previous work [31][32][33][34][35]. One may also consider them as "occurrences of motifs", "instances of motif", or "instances of patterns".

including a more rigorous treatment of the possible effect of outliers. These extended analysis results are reported in Sections 4.1 and 4.2, and summarized in Section 4.3. Furthermore, we report our experiment using the template proposed by Wohlin et al. [47] to ease readers to follow our work.

The main contribution of this paper is fourfold. First, our study is the first that investigates several human-related factors that may affect the effective use of design patterns in terms of the time taken to complete maintenance tasks correctly. Second, a list of the factors ranked according to their degree of significance of correlation with the time to correctly complete change tasks is identified. The ranked list could be a useful reference to project managers when allocating the tasks to maintainers with different amounts of work experience. Third, we identify that the presence of pattern-unaware solutions in a program affects the effective use of deployed design patterns to develop the solution. Fourth, through qualitative case studies, we uncover a scenario in which deployed design patterns are not utilized by maintainers for completion of the change task.

The rest of this paper is organized as follows. Section 2 briefly reviews the concepts of design patterns and their deployment. Section 3 defines the goal, planning, design and procedure of our empirical study. Section 4 describes how the empirical data were collected and presents our observations, case studies, statistical analysis, and interpretation of the results. Section 5 discusses the threats to validity of our study. Section 6 discusses related work and Section 7 concludes this paper.

2. Background

In this section, we present the background information about our experiment.

2.1. Design Patterns

A design pattern is a design solution to a category of recurring design problems, using a similar template that consists of a problem/solution pair and other information. In such a template, the problem part qualifies a situation to be handled by the solution part, which consists of structured prose and sketches to handle the situation, specifying how object-oriented classes (called *participants*) are structured and how objects collaborate. In the solution part, three features of a programming language [5][31] are typically applied:

(i) *Abstract interfaces*. They are defined to provide the functional signature for different participants of the design patterns to interact. As an abstract interface contains no implementation, using it to statically separate two *concrete participants* will confine the ripple effects of changes to one participant rather than distribute these effects to other participants of the same design pattern.

(ii) *Dynamic binding*. It defers the binding decision between participants until runtime.

(iii) *Subclassing*. Available choices of binding decisions are implemented as subclasses of the abstract interfaces.

For example, the *Factory Method* pattern solves the problem that the types of objects to be created are often hard-coded. Its solution is to define an abstract interface for creating an object, allowing different subclasses of the interface to decide which concrete classes to be instantiated.

2.2. Deployed Design Patterns

In the rest of this paper, we refer to an implementation instance of a design pattern in a program as a *deployed design pattern*. To deploy a design pattern in a target program, developers should complete at least four steps. First, they should formulate their design problems. Then, they should select the design pattern(s) that match the design problems on hand, followed by applying the static structure of the solution templates of the matched design patterns to the programs. Finally, they should implement the collaboration among objects as specified in the corresponding solution templates [33].

Consider a scenario of deploying some additional design patterns in *JHotDraw* [23]. Suppose that the developers of *JHotDraw* want to prepare for a potential change task called *Image*. To support *Image*, suppose that a new menu "ToolBar Options" is deemed necessary. This menu would support different sizes of the icons displayed on the buttons of the toolbar. However, the exact sizes of the icons are to be determined at the time the change task is implemented. To ease the potential program modifications for *Image*, two additional design patterns are deployed in *JHotDraw*. The *Factory Method* pattern is deployed to introduce polymorphic creation of different sizes of icons and the *Observer* pattern is deployed to define a one-to-many dependency between a toolbar option menu item and its corresponding size for a set of icons.

Figure 1 depicts the pattern participants and classes in *JHotDraw* relevant to *Image* both before and after the above-mentioned deployment of design patterns. After the above-mentioned pattern deployment, even though there are more classes (for example, *ButtonFactory* and *SmallButtonFactory* in Figure 1) in *JHotDraw*, a maintainer could add a new *ButtonFactory* subclass when *Image* is performed. This approach is simpler than trying to modify all subclasses of *Tool*. We refer to such an approach as a *pattern-aware approach* (and its corresponding solution a *pattern-aware solution*) because it utilizes the deployed design patterns as the major motifs to organize their solutions. An underlying assumption is that a pattern-aware

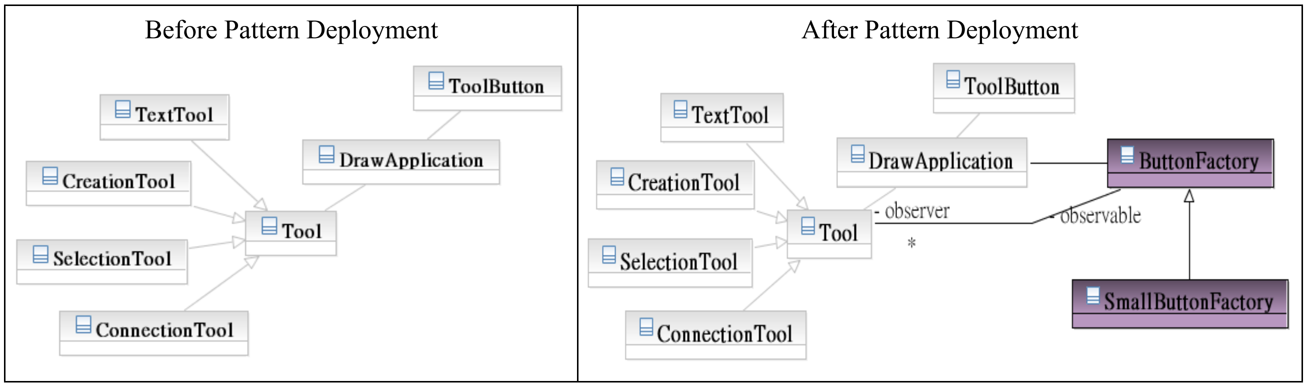


Figure 1. Relevant *JHotDraw* classes for the *Image* change task.

approach may result in a reduction of the effort for completing the change to this program in the future. If the assumption holds, we say that this pattern deployment *supports* the change task.

In general, multiple approaches to program modification may be adopted to complete a change task. We refer to a modification approach that does not rely on the ability of a deployed design pattern as an *alternative to (deployed) design patterns*, and its corresponding solution a *pattern-unaware solution*. For instance, to realize *Image*, a program modification approach that does not use the ability of the deployed design patterns does exist, which requires the modification of the classes *DrawApplication* and *ToolButton* only (see also Figure 15). In the next section, we describe an experiment to investigate various factors that may affect whether a pattern-aware solution or a pattern-unaware one is adopted.

3. Experiment

Following the presentation template proposed by Wohlin [47], we structure the reporting of our experiment to consist of the following five parts.

1) *Definition*: where we define our experiment in terms of problems and goals (Section 3.1),
2) *Planning*: where we discuss the design of the experiment (Section 3.2),
3) *Operation*: where we elaborate on how we carried out the experiment and collected the data (Sections 3.3 and 4.1),
4) *Analysis and interpretation*: where we present our analysis of the collected data and interpretation of the experimental results (Section 4.2), with references to threats to validity (Section 5),
5) *Discussions and conclusions*: where we discuss our results and their implications (Section 4.3), compare with related work (Section 6), and conclude the paper (Section 7).

3.1. Definition

In this section, we define the research problem, its context, and the goal of the study. Our primary goal is to identify individual human and program factors that contribute to the productivity (in terms of time spent) of maintainers in making correct software changes when they make changes to pattern-deployed programs. Specifically, we would like to find out the effects of these factors on change tasks that could, *but not necessarily*, be developed on top of the design patterns deployed in the program.

Finding the answers to this problem is important because many programs are deployed with design patterns, but maintainers may or may not use these design patterns in their implementation of change tasks. It is unclear what factors are in effect that correlate with maintainers' actions when they revise pattern-deployed programs. In this study, we intend to provide empirical data that contribute to further development of a richer theory which, as argued in Section 1, must take both relevant human and program factors into account.

In our study, the parameter of primary interest is the productivity demonstrated by each subject on program maintenance. We measure this parameter by the amount of time that a subject reports to have spent on *correctly completing the change task*. To understand the effect of each factor better, we analyze the data by compiling descriptive statistics from the perspective of individual human and programs factors and computing the correlations between each factor and the reported time for completion. We conduct our study in the context of time-constrained take-home coursework exercise setting, in which the human subjects are undergraduate and postgraduate students enrolled to a software engineering course of a university in Hong Kong.

We note that it could also be of interest to some readers to extend our investigation to the effects of factors on unsuccessful or incorrect attempts to a change task or on other parameters, such as the amount of faults present in the revised program. Readers may like to consider the following constraints in their studies (or replicate our experiments). First, owing to the time-constrained nature of a study context which is the same as ours, a subject (student) may submit his/her final but partially-completed program before a submission deadline. To be fair to all students for the same assignment, the submission deadline should not be selectively relaxed for a subset of subjects. On the one hand, granting a deadline extension to some but not all students may have adverse effects on those students who submit their programs on time. On the other hand, the amount of time needed to complete the outstanding parts of such a program is unknown and possibly indefinite.

Another consideration is that, although deploying a design pattern in a program by a developer may aim at easing some change tasks in the future, yet his/her intention of deploying the design pattern should not be confused with the intention of the maintainer who may not be the same person as the developer and may have different concerns. For instance, the assigned maintainer for a particular change task may complete the task by a method that he/she finds through searching an open-source repository or learns from friends or online forums, which are very popular in these days. Our efforts on studying human factors help to explore such issues. There are nonetheless many possible varieties. To balance between our resources and constraint, we choose not to deploy a complex experimental setting in this study.

We envisage that our results can be useful to three groups of stakeholders: researchers, practitioners, and students. Researchers may be interested in knowing whether a particular factor has a significant effect or a stronger correlation with the parameter of interest than some other factors. We therefore formulate a null hypothesis to validate this aspect and report the correlation results. Practitioners may want to know whether there are any specific data relevant to their own work scenarios. Hence, we compile the relevant descriptive statistics and discuss in detail the observations that may be of interest to practitioners. Students may like to acquire the skills to maintain software and study various kinds of approaches in revising a pattern-deployed program. We thus include sample solutions to assist students in the understanding of a set of possibilities in completing a change task.

In our experiment, we set out to study the following human and program factors that involve various aspects related to the change task, including the design patterns, program, programming language and subjects' work experience. Each factor is described in detail in Section 3.2.4.

Factor A. *Deployment of design patterns*

Factor B. *Presence of pattern-unaware solutions*[2]

Factor C. *Prior exposure to design patterns*

Factor D. *Prior exposure to the program*

Factor E. *Prior exposure to the programming language*

Factor F. *Prior work experience*

Considering the above six factors, a fully controlled experiment would require 2^6 experimental groups. Managing this number of experimental groups can be challenging. Constrained by both the budget and scope of our research projects, we did not conduct a full factorial design study. Rather, we study the correlation between the independent variables (the above six factors) and the dependent variable (time taken to correctly complete a change task). The *null hypothesis* of this experiment is as follows: *For each independent variable v, there is no significant correlation between v and the dependent variable.*

We intentionally set up this experiment in a course-based context, in which students were the subjects working on the change tasks as part of their coursework requirements. Based on the finding of this experiment, further experiments involving paid subjects from the software development industry may be conducted in a more cost-effective and manageable manner.

3.2. Planning

In the upcoming subsections, we shall introduce the subject program, *JHotDraw*, used in our experiment and the human subjects involved. Then we describe the change tasks and program versions of *JHotDraw* allocated to the subjects. Finally, we describe the factors under study.

3.2.1. Subject Program and Human Subjects

We use the *JHotDraw* (*version 6.0-beta2*) program [23] as testbed, as it has also been used in many research studies, both as illustrative example and for case study [2][4][34]. *JHotDraw* is an open-source drawing editor [23]. It has 14342 non-comment, non-blank lines of source code, distributed over 207 classes in 10 packages. A dozen design patterns was deployed in *JHotDraw*, including *Adaptor*, *Command*, *Composite*, *Decorator*, *Factory Method*, *Mediator*, *Observer*, *Prototype*, *Singleton*, *State*, *Strategy*, and *Template Method*.

[2] In the preliminary version of this paper, we use the term "Alternative to design patterns" to stand for this factor.

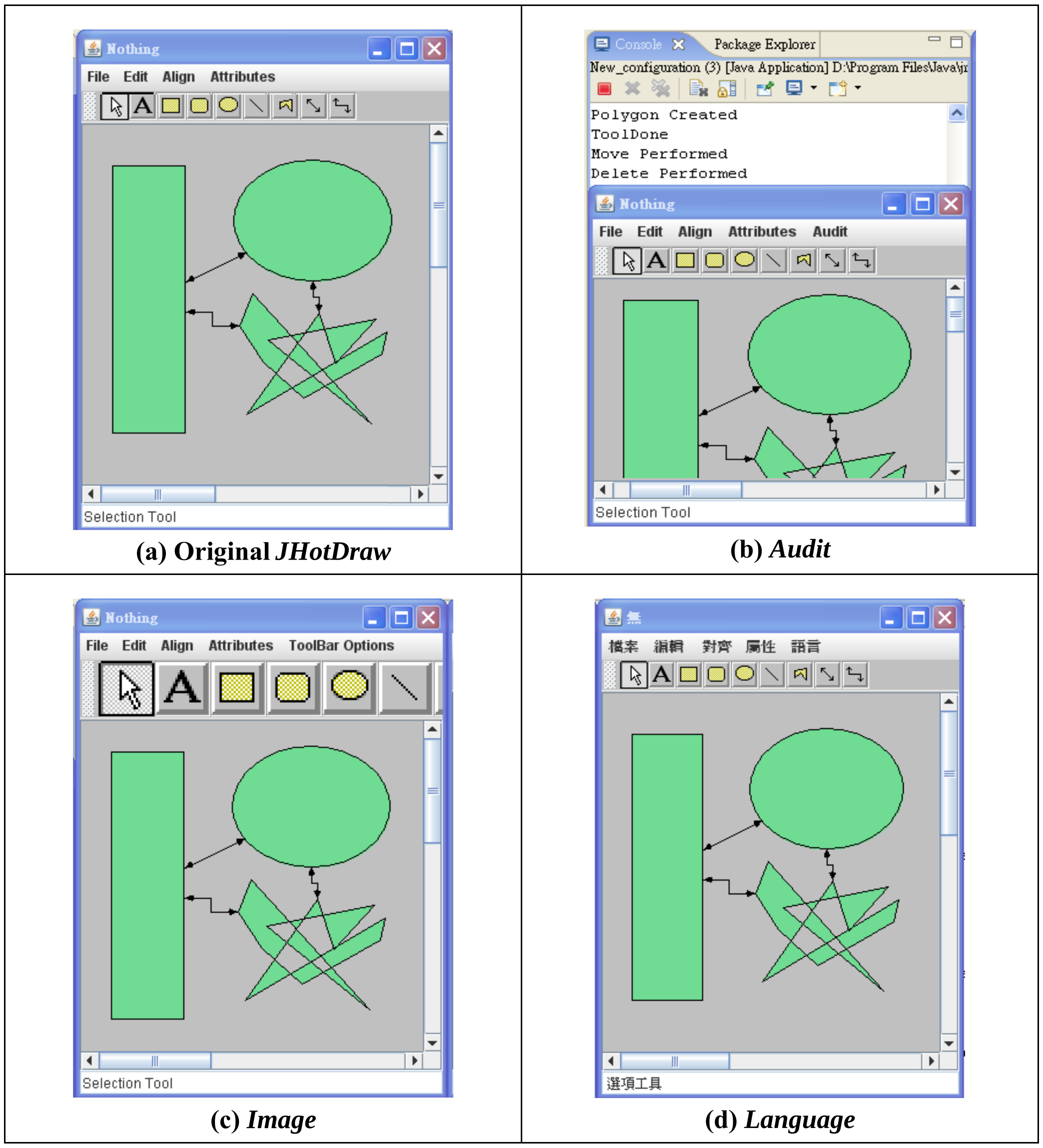
The human subjects belong to three class sections in software engineering courses offered in different semesters by a university in Hong Kong. Class sections 1, 2 and 3 consist of 63 postgraduate subjects, 31 undergraduate subjects and 24 undergraduate subjects, respectively. All undergraduate subjects were in their second year of study, who had complete freedom to enroll in either class section (section 2 or 3). There was no pre-screening policy to encourage or discourage any student to enroll in any of the two class sections. Unlike the undergraduate subjects who possessed no work experience at the time of the experiment, most of the postgraduate subjects were full-time software developers with at least one year's work experience in the industry in Hong Kong.

3.2.2. Change Tasks

Our study involves the completion of *three* change tasks on *JHotDraw*. To ease our reference, they are referred to as *Audit*, *Image*, and *Language*. Figure 2(a) shows a sample screenshot of the original *JHotDraw*, while Figure 2(b)–(d) shows a corresponding screenshot of the program after the completion of each of these three tasks *Audit*, *Image*, and *Language*, respectively. Details of the change tasks are described as follows.

(i) *Audit*. This change task specifies adding a new menu "Audit" to the subject program with a menu item "Record Action History". When the menu item is enabled, log messages of the following actions will be displayed on the console:

- A *figure* (such as text, rectangle, round rectangle, ellipse, line, or polygon) is created on a canvas.



```
Polygon Created
ToolDone
Move Performed
Delete Performed
```

(a) Original *JHotDraw*

(b) *Audit*

(c) *Image*

(d) *Language*

Figure 2. Sample screenshots of *JHotDraw* before and after the completion of each of the three change tasks.

- A *connection* between figures is created on a canvas.
- A created figure or connection is *moved* from one location on the canvas to another (possibly the same) location on the same canvas.
- A GUI *command* (such as cut, paste, duplicate, delete, group, ungroup, send to back, bring to front, undo, redo, align and change attribute) is issued.

Figure 2(b) shows a sample screenshot of *JHotDraw* after the completion of the *Audit* change task, where some log messages are displayed on the console corresponding to the actions of the user.

(ii) *Image*. This change task specifies adding a new menu "ToolBar Options" to the subject program. The new menu consists of two menu items, "Small" and "Large", which determine the set of icons displayed on the buttons of the toolbar. Two given sets of icons were provided, whose appearance is the same, except that every icon in the first set is a scaled-down version of a corresponding icon in the second set. Figure 2(c) shows a sample screenshot of *JHotDraw* after the completion of the *Image* change task, where the "Large" set of icons are selected and displayed instead of the original "Small" set of icons displayed in Figure 2(a).

(iii) *Language*. This change task specifies adding a new menu "Language" to the subject program. The new menu supports the choice of a language option from three available: "English", "Traditional Chinese", and "Simplified Chinese". All texts appearing in the following components will be displayed in the chosen language: the file name of the picture, the menu bar, the tool tip text displayed in the toolbar, and the status bar. Figure 2(d) shows a sample screenshot of *JHotDraw* after the completion of the *Language* change task, where the "Traditional Chinese" option is selected and both the menu bar and status bar are displayed in traditional Chinese characters instead of English.

3.2.3. Program Versions

As explained in Section 3.1, our primary goal is to identify factors that individually affect the maintenance of pattern-deployed programs. According to the documentation that came with the distribution of the *JHotDraw* program version (which we call the *original version*) we obtained from Sourceforge, 12 design patterns had been deployed. However, none of these design patterns had been deployed for the purpose of supporting the three change tasks described in Section 3.2.2 above.

Therefore, we refactored *JHotDraw* into three different *program versions* called Version-A, Version-I and Version-L. For ease of reference, we also refer to them simply as the *refactored versions*.

- Version-A is the result of deploying additional design pattern(s) to the original version to support *Audit*, but not *Image* or *Language*.
- Similarly, Version-I (respectively Version-L) is the result of deploying additional design pattern(s) to the original version to support *Image* (respectively *Language*) but not other change tasks.

Table 1 summarizes the allocation of program versions to subjects (students) of different class sections. Specifically, subjects of class sections 1, 2, and 3 were asked to work on Version-A, Version-I, and Version-L, respectively, to implement all the three change tasks in the order specified in Table 1.

It is infeasible in our context to get a large pool of human subjects having the same background. Under this constraint, we allocate different change tasks among the class sections in the following way. First, by exhaustion, there are six possible sequences of the three change tasks (*Audit*, *Image*, and *Language*). Every such sequence s can be combined with every program version c, to form 18 pairs $\langle s, c\rangle$. We randomly assign a unique number from the set {1, 2, 3} to each program

Table 1. Versions of *JHotDraw* assigned to subjects.

Class section	Number of subjects	Version of *JHotDraw* assigned to subjects of the class section	Assigned order of change tasks
1	63	Version-A (with deployed design patterns to support *Audit* only)	1. Image 2. Audit 3. Language
2	31	Version-I (with deployed design patterns to support *Image* only)	1. Audit 2. Language 3. Image
3	24	Version-L (with deployed design patterns to support *Language* only)	1. Language 2. Audit 3. Image

version, and then randomly select three out of the 18 pairs ⟨*s*, *c*⟩ such that for each selected pair, the number assigned to *c* is also the position of the change task that *c* is refactored for. Based on this method, we assign 1 to *Language*, 2 to *Audit*, and 3 to *Image*, and have identified the three orders of the three tasks as shown in Table 1. Finally, we randomly assign the three selected pairs to the three class sections. In summary, the mapping among subjects, versions and order of change tasks was designed to ensure that the subjects of different class sections performed a change task with support by its corresponding deployed designed patterns and the other change tasks without the support of deployed design patterns, and that they would be at different levels (0 or 1) of prior exposure to the program (Factor D, as explained in Section 3.2.4) when completing different change tasks.

Some readers may like to use the original version of *JHotDraw* as a baseline version and assign a group of subjects to implement the change tasks in this baseline version. We did not choose such a setting for our experimental design as it would not fit our context. Specifically, Factor B (presence of pattern-unaware solutions) requires that a program must be deployed with design pattern(s) to support change tasks. Otherwise, we cannot study the scenario of subjects using the corresponding design patterns in their (pattern-aware) solutions, or not using it despite their presence (to produce a pattern-unaware solution). We were also constrained by the availability of only three class sections, each must be given the *same* assignment because it is a part of their coursework assessment. A fourth class section of students was not available for us to assign the original version of *JHotDraw* to them to work with.

The rationale and procedure of preparing these refactored program versions are now elaborated as follows. Different design patterns may be deployed to remove different bad smells of code associated with a particular change task. We firstly inspected the code of the original version and used our expert judgment to determine the regions of code that could be modified to produce a workable version for each change task. We then followed the refactoring procedures described in [24] to spot the bad smells in this code region and removed them by producing code-refactored versions of the *JHotDraw* program. In so doing, we did not pay attention to whether a removal of code smells and the chosen refactoring templates may help a particular change task. Rather, we followed the code of professional ethics to do our best to remove code smells and selected the best templates we could find when performing the code refactoring process. Table 2 summarizes the bad smells and refactoring procedures that we applied to the original version of *JHotDraw*

We however note that design is a creative activity, and different designers may have different solutions. A solution accepted by one designer may be rejected by another designer..

To validate whether the refactoring procedure preserves the functionality of the programs, we performed category-partition testing. After performing a category-partition analysis, we tested the programs with inputs selected to exhaust all feasible category-choice combinations, using the original version of *JHotDraw* as the test oracle. Code inspection was also done to verify that all the deployed design patterns conform to the structure and collaboration in their original specifications [16].

We recall that each refactored version contains 12 originally deployed design patterns plus the one that we introduced. The original version has 14342 lines of code. Our count shows that each refactored version contains no more than 100 additional lines of code. Our focus is on whether the deployed design patterns were used or not used in a solution. We have worked out our own solutions, in which the amounts of code modification were in the range of 1000–3000 lines of code. Indeed, as shown in Table 1, in *none* of the refactored versions was code added to support *all* the three change tasks, yet each subject was required to complete the three tasks. Thus, using our own solution as a reference, the changes we made in refactoring the programs constitute no more than 100 lines of code out of 1000–3000, which amount to around 3–10% for solutions that adopt deployed design patterns, and 0% for any other solutions.

3.2.4. Independent Variables (Factors)

The intuition on the relationships among the six factors under study is depicted in Figure 3. These factors correspond to the independent variables of our study. Readers may observe that there are factors related to the program (Factor A and Factor B).

Table 2. Procedures used to refactor *JHotDraw*.

Change	Refactoring procedures
Audit	• Extract special case logic* into *Decorator*
Image	• Replace hard-coded notifications* with *Observer* • Introduce polymorphic creation* with *Factory Method*
Language	• Replace hard-coded notifications* with *Observer* • Replace one/many distinctions* with *Composite*

* The bad smells removed

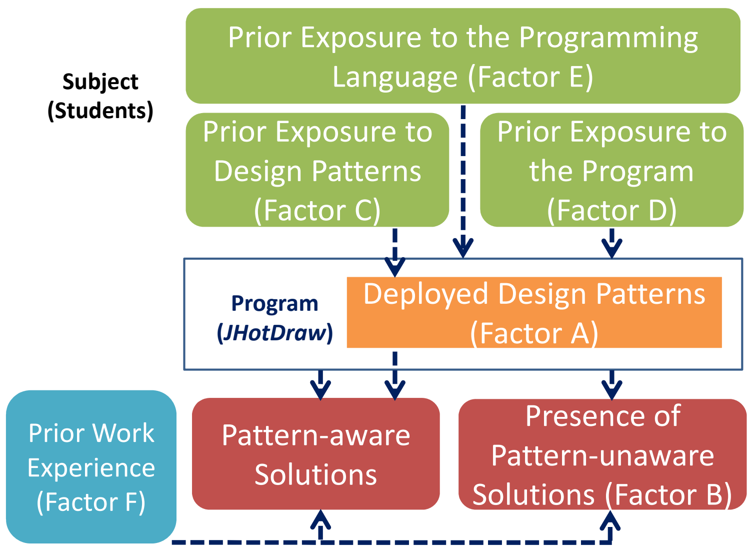

Subject (Students)
Prior Exposure to the Programming Language (Factor E)
Prior Exposure to Design Patterns (Factor C)
Prior Exposure to the Program (Factor D)
Program (JHotDraw)
Deployed Design Patterns (Factor A)
Prior Work Experience (Factor F)
Pattern-aware Solutions
Presence of Pattern-unaware Solutions (Factor B)

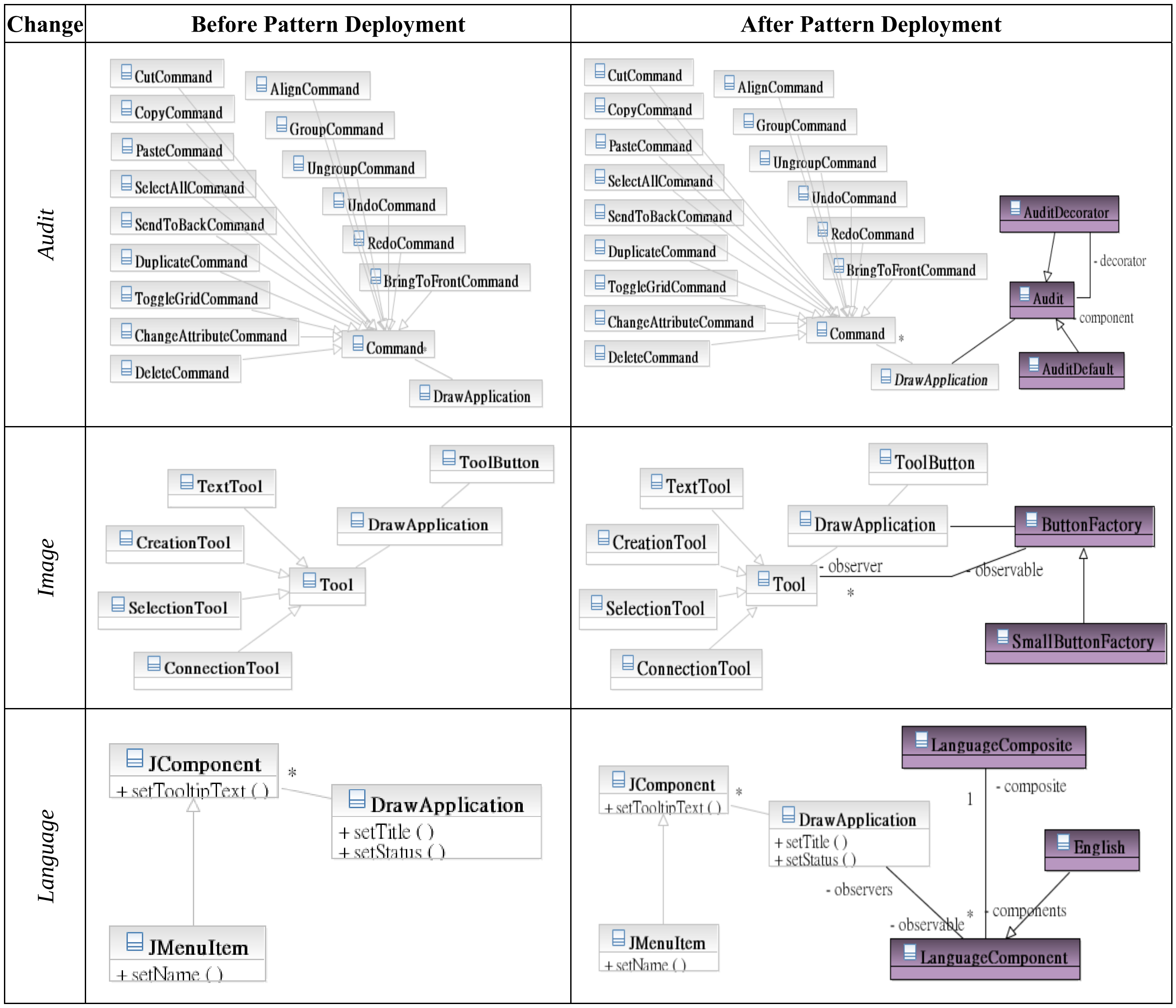
Figure 3. Intuitive relationships among Factors A–F.

To implement solutions for the change tasks, the maintainer needs to possess certain types of knowledge that are directly relevant to these two factors. Thus, we consider that these types of knowledge are related to the subject's familiarity with design patterns (and their deployment) as well as the class structure (plus other details) of the program. Consequently, we include in our study Factor C (Prior exposure to design patterns) and Factor D (Prior exposure to the program), respectively. Moreover, we consider that Factor C and Factor D both demand certain levels of familiarity with the programming language (Factor E) when the subject works on the program code. Finally, we also include Factor F (Prior work experience) in our study, as developers/maintainers may benefit from their experience and transfer their practical skills and knowledge across different domains.

Factor A. *Deployment of design patterns*. This factor refers to whether deployed design patterns are present to support a change task. By support, we mean that a solution uses a deployed design pattern to coordinate the objects of the classes involved in the design pattern that implements the solution. In the experiment, there are two levels for this factor, 0 or 1, corresponding to the absence or presence of such deployed design patterns in the program, respectively. As Version-A was deployed with design pattern(s) to support *Audit*, the value of Factor A is 1 when a subject works on it to implement *Audit*. On the other hand, as Version-A contains no deployed pattern(s) to support *Image* or *Language*, the value of Factor A is 0 when a subject works on it to implement these two change tasks. The values of Factor A are assigned in a similar way when subjects work on other program versions and change tasks.

Figure 4 shows the relevant *JHotDraw* classes for various change tasks before and after refactoring. In the figure, classes that were added to *JHotDraw* during the refactoring process for a change task are shaded in purple. Indeed, after refactoring, the resultant *JHotDraw* version becomes larger in size due to the introduction of additional classes. Refer to the texts and figures in Section 4.2.2 for more detailed discussions.

Factor B. *Presence of pattern-unaware solutions*. As explained in the last paragraph of Section 2.2, even with deployed design patterns to support a change task, a maintainer may not utilize them in their implementation of the change task. Instead, he/she may seek an alternative solution to complete the change task. Obviously, the existence of such an alternative is a factor that could influence the effective use of deployed design patterns. In our study, we consider that this factor may be manifested in three levels. Level 0 means that the only obvious program modification approach to complete the change task is to utilize any deployed design patterns in the program, and other alternative approaches would need to revise the code much more extensively. An example change task of Level 0 is *Audit*, where there is no clear alternative (other than utilizing the deployed design patterns) for completing the change task. Level 1 means that such an alternative exists, and the effort required by this alternative method is comparable to the approach of utilizing the deployed design patterns. *Language* is an example change task of Level 1. Level 2 means that there is an alternative program modification method, which in fact produces a shortcut solution that requires significantly less effort to complete the change task than using the deployed design patterns. *Image* is a change task of Level 2. We thus investigated how and to what extent the presence of these different alternative (pattern-



Note: Classes added to *JHotDraw* during the refactoring process for a change task are shaded in purple.

Figure 4. Relevant *JHotDraw* classes for various change tasks before and after refactoring.

unaware) solutions would influence the effective use of deployed design patterns. We use our own solutions and expert judgment to define these levels prior to the execution of the experiment.

Factor C. *Prior exposure to design patterns*. In general, we expect that a subject who has spent more time learning design patterns will probably perform better when working with programs in which design patterns have been deployed [17]. Thus, apart from the deployment of design patterns, we took into account the amount of time that a subject had previously spent on learning design patterns. In this experiment, the subjects were required to complete a survey in a tutorial session, where they reported when they first learned design patterns. The survey shows that the subjects have zero to nine years of prior exposure to design patterns.

Factor D. *Prior exposure to the program*. A subject may find it easier to complete the change of a program with which he/she is familiar. In our study, *JHotDraw* is the testbed and subjects having prior exposure to *JHotDraw* are expected to be more familiar with its specific details, such as its internal structure.

We consider only two levels for this factor, 1 and 0, representing *yes* (that is, with experience in modifying the implementation of *JHotDraw*) and *no* (otherwise), respectively. Intuitively, at Level 1, a subject (who has been exposed to the program before, typically through previous hands-on programming experience on a different change task) may have an advantage over the same subject at Level 0 (who works on the first change task and, hence, encounters *JHotDraw* the first time). Initially, all subjects were of Level 0, as they had no experience in modifying *JHotDraw* prior to the study. According to the setup of our experiment, each subject had to sequentially complete three change tasks. As soon as a subject completed the first task, he/she had gained hands-on programming exposure to the program and, hence, would be "elevated" to Level 1. Thus, each subject was at Level 0 when working on the first assigned change task, and at Level 1 when working on the next two change tasks.

Factor E. *Prior exposure to the programming language*. Generally, assuming that other factors are held the same, a person who has learned a programming language for a longer time is expected to be more proficient in the language and, hence, perform better in maintaining programs written in the language. Thus, one potentially relevant factor could be the amount of time a subject had spent on learning Java, the programming language in which *JHotDraw* is written. Similar to Factor C, in the same survey, the subjects also reported when they first learned Java, from which the data values for Factor E were calculated.

Factor F. *Prior work experience*. To consider the effect of the subjects' prior work experience, we took into account the amount of time that a subject had worked in the IT industry. Intuitively, we expect a software professional with more work experience to possess higher capability and, hence, need to spend less time to complete a maintenance task. Again, the subjects reported the amount of their work experience (in months) in the same survey from which data for Factors C and E were collected.

3.3. Operation – Experimental Procedure

To prepare the subjects for the task assignments, we conducted a three-hour pre-experiment tutorial session for each class section to explain to them the functional requirements of the three change tasks under study. *Eclipse* was used as the software development environment by all subjects.

All subjects attended at least one three-hour tutorial session, as shown by the attendance sheets that we collected. In the tutorial session, we spent most of the time explaining the functional requirements of the change tasks and demonstrated how to use *Eclipse* to open, close, compile, and run a *JHotDraw* project. We did not describe *JHotDraw* in detail but walked through the big picture about the general structure of *JHotDraw* using the API documentation and class hierarchies bundled in its original version. We mentioned how the API documentations were organized, scrolled down the browser quickly to show that each class has descriptions of its fields and attributes, and indicated how to interpret classes that form part of a class hierarchy. We also conducted a survey to collect the data for Factors C, E, and F from the subjects during each tutorial session. The survey also indicates that no subject had any prior exposure to *JHotDraw* before this experiment.

To maintain the validity of the experiment, we did not mention any design patterns (including the addition of design patterns via refactoring) to any subjects, as we did not want to consciously bias the experiment by hinting the subjects with specific points of interests in the program that would help complete the change tasks. We spent most of the time explaining the functional requirements of the change tasks and demonstrated how to operate *Eclipse*. We also demonstrated how to use *System.out* of Java API to output a Chinese character and how to extend the original version of *JHotDraw* by adding a dummy menu item through a simple and dummy program. An instance of the tutorial content is available online [44]. In the tutorial sessions, we instructed the subjects to mark down the time durations whenever *Eclipse* was used for each change task and then report the total of these time durations (in minutes) spent on each task.

After the tutorial session, each subject was given a sequence of three assignments as shown in Table 1, which was also mentioned in the tutorial website [44] so that readers may replicate our experiment. Each assignment consists of a specific change task and a particular version of the subject program. For instance, the subjects of class section 1 were given Version-A and asked to complete the change tasks *Image, Audit*, and *Language*, one at a time, in that order. Along with the release of an assignment to each class section, we also provided a binary version of our solution to the corresponding change task. The binary version had been obfuscated (and regression tested) so that decompiling the binary code back to Java source would not produce comprehensible code. Subjects were asked to complete a change task so that the functional behavior of their submitted versions was the same as that of the provided binary version. The exact wording used in the tutorial and the assignment system are shown in the tutorial website [44].

The following additional measures were exercised when administering the task assignments.

- Each subject was free to use any program modification approach.
- Each task assignment was performed by individual subjects.
- Each subject was given one month to complete each task. After the submission deadline of the first task assignment, we distributed the second task assignment to each subject. Similarly, after the submission deadline of the second task assignment, we distributed the third one.

- The subjects were required to complete each change task by modifying the given program version, not by modifying the program version that they submitted in earlier task assignments, if any.
- Each subject was required to report the time (in minutes) actually spent on the assigned task. The subjects were clearly assured that the reported amount of time spent would not affect the scores they received from the assignment.

Finally, as the experiment involved human subjects, we have ensured that it was conducted in a way that conformed to the safety and ethical requirements of the university involved as well as those of the research grant sponsors of this work.

4. Results and analysis

This section presents the data collected during the experiment and relates the performance of the subjects (in terms of the time spent on correctly completing the change tasks) with the six factors under study. The term "factor" here refers to human or program factor, which corresponds to a "variable" in statistical analysis.

4.1. Operation – Data Collection

Recall that there were 118 human subjects in total and each of them was asked to perform three change tasks. Thus, we had expected to receive 354 program versions produced by the subjects. However, after the deadlines, there were 4 missing submissions, resulting in a total of 350 program versions actually collected.

To test the functional correctness of the program submissions, we followed the same procedure as done by Ng et al. [34] as closely as possible. We first performed category-partition analysis and then selected test inputs to exhaust all feasible category-choice combinations. For each received program version, if all test cases were passed, we considered that the change task had been correctly completed. One of the authors, Ng, served as the human oracle to judge whether the revised program versions behave correctly on the test runs. A more detailed description regarding our category-partition analysis can be found in the work by Ng et al. [34].

After testing, we obtained 165 correct program submissions for analysis. We further observe that there is a particular program submission for which the reported time of completion is 13.8 times the mean maintenance time of the group, and is about 300% longer than the second longest completion time to produce a correctly changed program. The submission can also be visually revealed from Figure 5, in which the maintenance time taken is plotted against the program submissions, ranked in ascending order of the time taken for the change task. We considered this submission, indicated by the topmost (and rightmost) data point in Figure 5, as an anomalous case and, hence, excluded it from analysis. Therefore, the *whole data set* consists of the remaining 164 correct program submissions, and it forms the basis of our observations and statistical analysis. (We shall further consider two subsets of the whole data set for statistical analysis later in Section 4.2.3).

4.2. Analysis and Interpretation

For each of the 164 program submissions in the whole data set, we studied the relationship between the (reported) maintenance time spent by the subjects and the six potential factors under study. In what follows, we first report our observations and detailed analysis for each factor in Section 4.2.1. Afterwards, Section 4.2.2 presents some sample case studies where we discuss our observations of the program modification approaches adopted by the subjects. Section 4.2.3 reports our statistical analysis of the significance and correlations of the six factors under study. Finally, Section 4.3 summarizes our experimental results and analysis.

4.2.1. Observations on Indiviudual Factors

Factor A. *Deployment of design patterns.* We find that the mean, median and standard deviation of the maintenance time

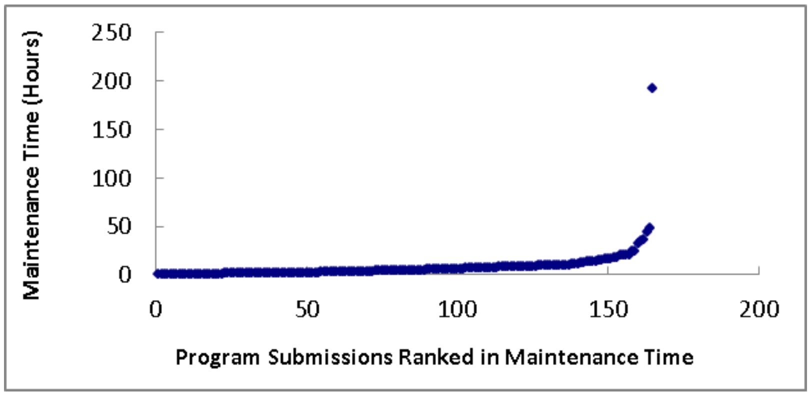


Figure 5. Scatterplot of program submissions in order of the maintenance time taken.

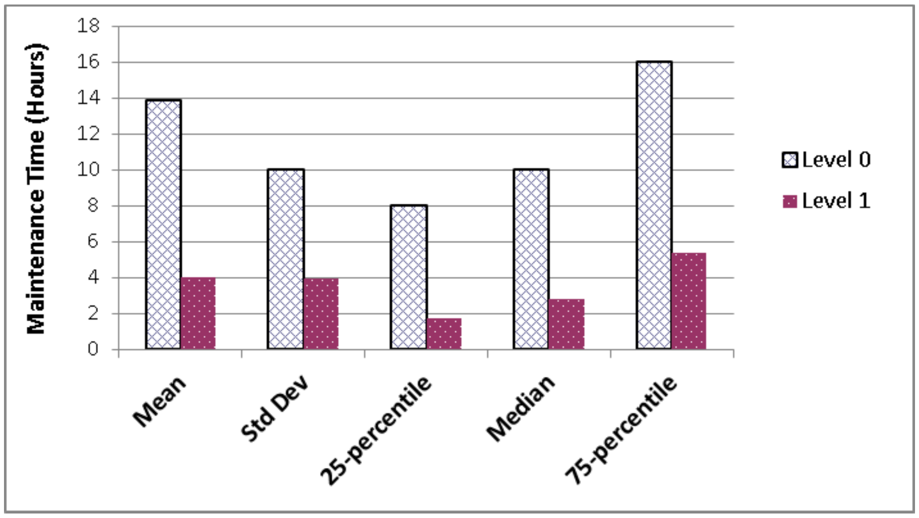


Figure 6. Statistics of time taken versus different levels of deployment of design patterns (Factor A).

required to complete a change task for Level 0 are 13.8, 10.0, and 10.0 hours, respectively, and the corresponding values for Level 1 are 4.0, 2.8, and 3.9 hours, respectively (see Figure 6 for a graphical view of these and other statistics). The result indicates that, on average, a subject would take around 3.5 folds (in terms of both mean and median) of maintenance time to complete a change on a program version at Level 0 (*without* deployed design patterns to support the change task) when compared with that involving a program version at Level 1 (*with* deployed design patterns to support the change task). We also find from Figure 6 that even when we compare the time spent at the best 75-percentile for programs at Level 1 with that at the best 25-percentile of programs at Level 0, maintenance of a pattern-deployed program is still much faster.

In terms of standard deviation, Level 0 is about 2.6 folds of Level 1. It also shows that making a change based on a pattern that supports the change can make the maintenance process more predictable in terms of time spent.

It is interesting to observe that the standard deviation at either level is relatively large. For Level 0, the magnitude of the standard deviation is almost the same as the median value. Similarly, for Level 1, the standard deviation is almost the same as the mean value of the same level. In other words, although the time needed to perform maintenance is shortened by performing the change on top of a pattern that is already deployed to support the change task, yet there are other factors that may cause the maintenance process to become not quite predictable. This calls for a need to conduct more studies on human-related factors in the future.

Let us consider the following hypothetical scenario. Suppose that at Level 1, up to 50% of the mean maintenance time (that is, 2 hours) is spent on finding whether any deployed design pattern may be used to support the current change task. Suppose that, however, there is in fact unfortunately no such deployed design pattern (that is, the program version is actually at Level 0 of Factor A). Then, on average, the overhead of looking for deployed design patterns is only about 14.5% (= 2.0 / 13.8). This overhead is significantly smaller than the measured variation (i.e., the standard deviation), which is 72.5% of the mean value. It seems that, on average, it may be a good idea to search for usable deployed patterns before looking for alternatives.

Factor B. *Presence of pattern-unaware solutions*. For this factor, we primarily focus on the completion of change from a refactored version, because subjects may (unintentionally) adopt either the pattern-aware solution or the pattern-unaware solution. We have set up three levels for Factor B. To determine whether a program falls within a particular level, there is no objective criterion (or else, we should have developed the ground theory rather than working on an empirical study to find out whether Factor B could be an issue in using design patterns for completing a change task). In the study, we used expert judgment. Specifically, we inspected the code of each version and determined whether it used a design pattern and its dynamic mechanism to implement a required change task. This part is relatively straightforward. In our case, basically, we simply check whether a program may have modified or extended the classes in a deployed design pattern. By doing so, we have already sorted out a large portion of the submitted versions that should have used an alternative (pattern-unaware) solution. If the program did not exhibit this feature, we used our development experience and expert knowledge to assign a level of Factor B to the program.

Figure 7 shows the ratio of the number of pattern-aware solutions to pattern-unaware solutions. (Refer to the case studies in Section 3.2.3 for a more detailed analysis of the pattern-aware solution and pattern-unaware solution for each change task.)

First, for a program version at Level 0, our data show that all correctly completed solutions were implemented as pattern-aware solutions. (We note that some subjects implemented pattern-unaware solutions but none of these implementations

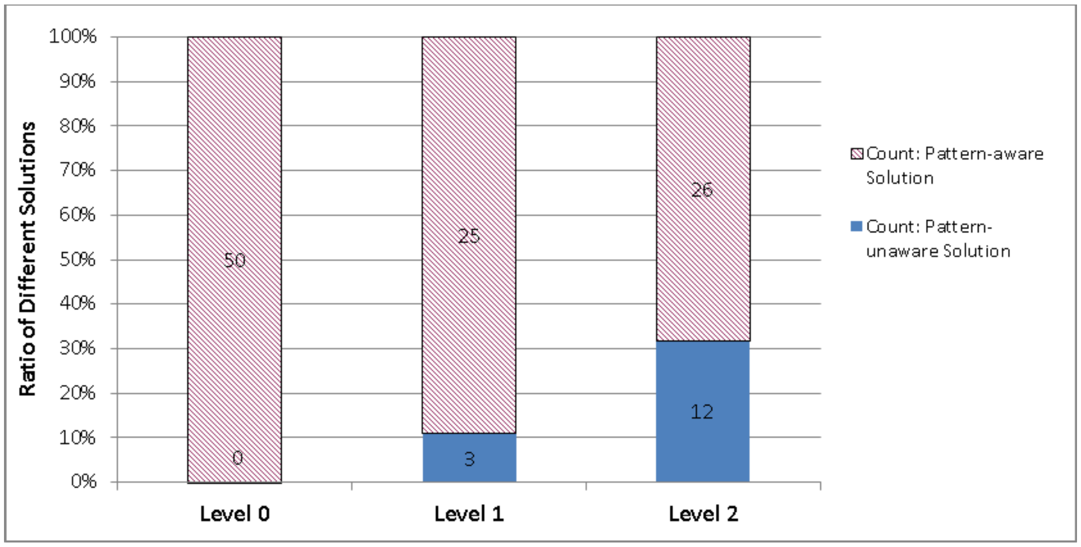


Figure 7. Percentage of (correct) programs implementing different solution approaches (at different levels of Factor B).

passed our pre-developed test suite.) For Levels 1 and 2, we observe that the number of pattern-aware solutions is much higher than that of pattern-unaware solutions. Indeed, for Level 2, even though a shortcut solution exists, the number of pattern-aware solutions is still more than double that of pattern-unaware solutions. Thus, the result suggests a clear tendency of the subjects in making use of the deployed design patterns that support the change task, even when these design patterns are not strictly necessary.

Figure 8 shows the mean and standard deviation of the time taken to complete a change by implementing a pattern-aware or pattern-unaware solution from a refactored version. We observe that implementing a pattern-aware solution is more efficient (than otherwise) in terms of mean maintenance time. However, we find that the standard deviations for pattern-aware solution are of almost the same scale as those of their corresponding means, whereas the standard deviations for the pattern-unaware solutions are much smaller than their means.

In particular, for Level 2, the means for the pattern-unaware and pattern-aware solutions are 4.7 and 3.9 hours, respectively, and their standard deviations are 3.0 and 4.6 hours, respectively. This result is interesting. We recall that shortcut (pattern-unaware) solutions exist at Level 2, which should require less effort to complete the change tasks. The empirical result,

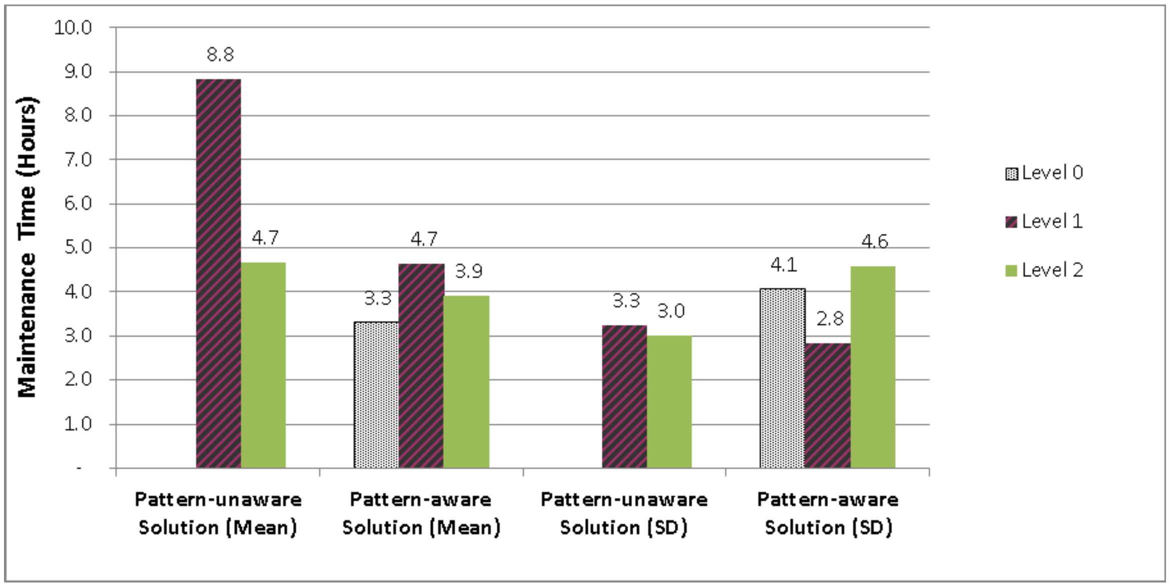


Figure 8. Time taken to implement different solution approaches (at different levels of Factor B).

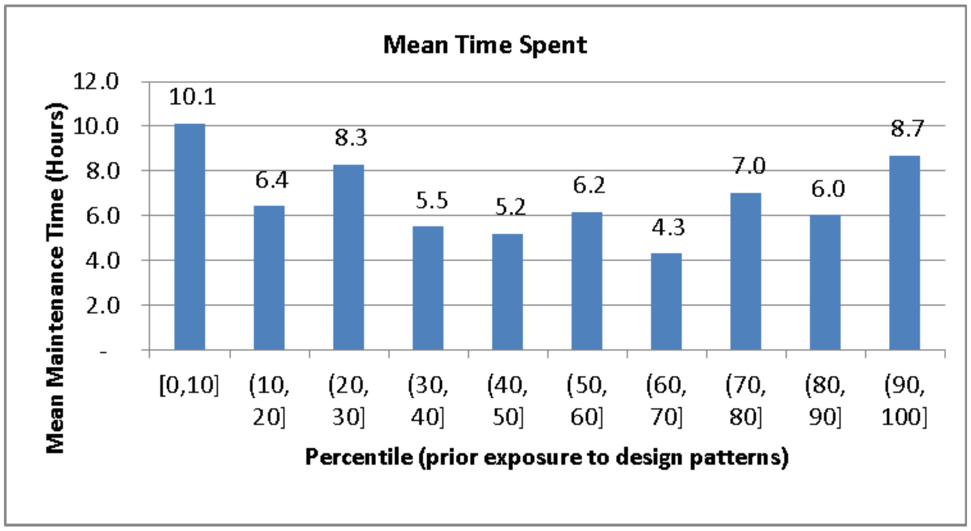


Figure 9. Mean time taken by subjects of different percentiles of prior exposure to design patterns (Factor C).

however, shows that the time taken to implement a pattern-unaware solution is actually longer on average, but has a lower variation. In our development experiences, we conjecture that the use of a shortcut (pattern-unaware) solution instead of the "more obvious" (pattern-aware) solution requires a certain level of competence in conceiving a different way of solution. Although an in-depth analysis on the underlying reasons falls beyond the scope of the current study, it is clear that the rationales of making such decisions warrant further research.

Factor C. *Prior exposure to design patterns*. Figure 9 shows the mean time taken by the ten different percentiles (each shown as a bar) of prior exposure to design patterns. Consider the leftmost bar, for example. It represents the one-tenth of the subjects who possess the least prior exposure to design patterns. The second leftmost bar represents the subjects who possess the second lowest one-tenth prior exposure to design patterns. Other bars can be interpreted similarly. The y-axis represents the mean amount of time taken to complete a change task by the percentile group. Figure 10 shows the corresponding lower boundary of prior exposure to design patterns in terms of hours. For example, the second leftmost bar shows that the lower boundary for second one-tenth of the subject is 13 hours. Other bars can be interpreted similarly. The y-axis can be interpreted in a similar way as that of Figure 9. Figure 11 shows the standard deviation of the time taken by the subjects in different percentile groups in terms of the amount of prior exposure to design patterns. Again, the x-axis and y-axis can be interpreted in a similar way as those of Figure 9.

We observe from Figure 9 that the height of the leftmost bar is 10.1, which is the highest among all bars. All other bars are at least 13.9% lower than this bar. The height of the shortest bar (that is, the seventh bar counting from the left) is only 42.5%

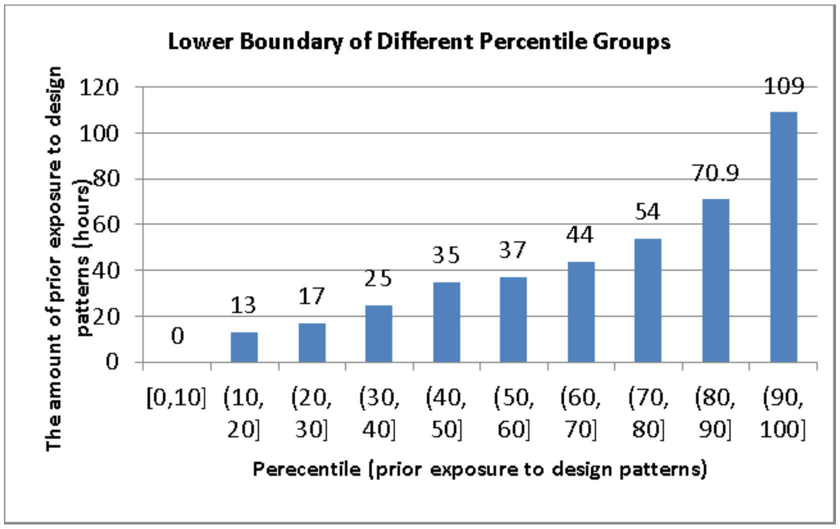


Figure 10. The lower boundaries of different percentiles of prior exposure to design patterns (Factor C).

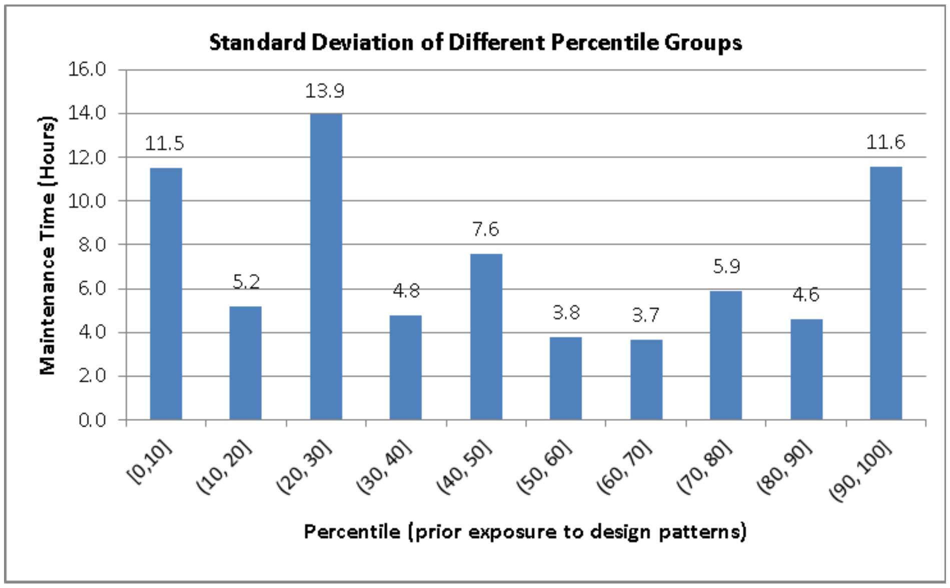


Figure 11. Standard deviation of time taken by subjects of different percentiles of prior exposure to design patterns (Factor C).

of that of the leftmost bar. Owing to the nature of empirical study, the exact values may not be the same for another experiment. The current result indicates that, on average, even a small amount of exposure (which is 13 hours as seen from Figure 10) to design patterns can help shorten the maintenance time. We also analyze the medians of these groups. We find that the median of the best group (a.k.a. the shortest bar) is about 45.9% of that of the leftmost bar and the leftmost bar is still the longest one.

Overall, the bars in Figure 9 from left to right exhibit a trend which initially drops as the amount of prior exposure to design patterns increases, and then rises again. In terms of the mean time spent, the seventh bar counting from the left (that is, the shortest bar) is only 82.6% of the second shortest bar. This difference is noticeable. However, in terms of the median time spent, we do not observe a clear trend among the percentile groups.

From Figure 10, the cutoff (threshold) value for the shortest bar of Figure 9 is 54 (hours), which is slightly more than one week. In terms of median time spent, we find that the shortest bar is the third leftmost one and its cutoff value is 17 (hours). The result indicates that even a limited amount of the subjects' exposure to design patterns can be very useful.

Factor D. *Prior exposure to the program*. We find that the subjects of 65.4% of (correct) submissions have prior exposure to the program. (We note that although in Table 1 we have described the mapping between versions and the subjects, yet whether a modified version represents a successful completion of task cannot be predicted in advance.) A further analysis shows that on average, they could complete a change task in 5.7 hours with a standard deviation of 8.0 hours. On the other hand, for those without prior exposure to the program, 9.2 hours are needed on average and the corresponding standard deviation is 7.3 hours. Compared with the mean, the standard deviation in either case is relatively large.

The ratio of mean maintenance time between the two groups is 160% (= 9.2 / 5.7). The result echoes the common practice that a work supervisor tends to find a subordinate who is familiar with the program to perform its maintenance. Another interpretation of the result is that it is more effective to familiarize a maintainer with the program that he/she will handle in the future. On the other hand, we observe that the standard deviations of the two groups for this factor are about the same. It may indicate that there are some complications in terms of the concept of prior exposure to the program. Further work may help iron out the dominant sub-factors of this factor.

Factor E. *Prior exposure to the programming language*. Figure 13 shows the mean maintenance time for groups with different amounts of prior exposure to the programming language. The x-axis and y-axis of this graph can be interpreted in a similar way as those of Figure 9, except that in Figure 13 the 10 percentile groups are categorized according to the amount of prior exposure to the programming language.

Here we do not observe a clear trend from left to right. However, in some cases, some exposure to the programming language does help shorten the maintenance time. It provides a piece of evidence to task assignment personnel that prior knowledge of the language, once sufficient, may not be a dominant factor to enable more effective maintenance.

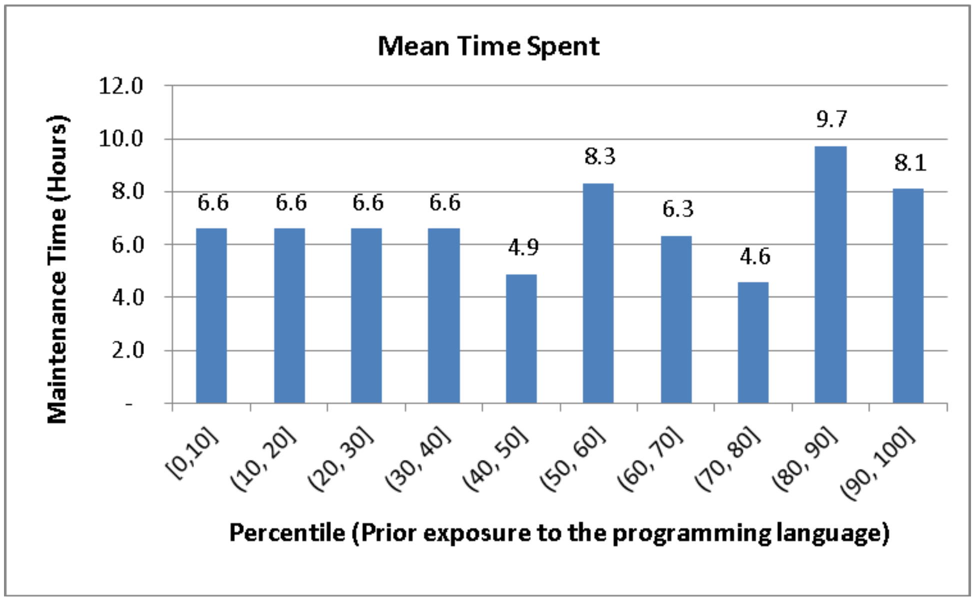


Figure 13. Mean time taken by subjects of different percentiles of prior exposure to the programming language (Factor E).

Factor F. *Prior work experience*. Figure 12 shows a scatterplot of maintenance time versus prior work experience of the subjects. The data on this factor were also collected via a survey. We observe that the vast majority of the data points reside in the lower part of the plot. That is, the maintenance time spent is not high for most of the subjects across a wide range of prior work experience. In this connection, prior work experience does not appear to have a dominating effect on the performance of the maintainer.

At the same time, we also notice that the upper right region of the graph is practically empty, indicating that experienced subjects rarely have to spend a great deal of time in completing the maintenance task correctly. In other words, maintainers who possess more work experience tend to be able to complete a change task without taking an amount of time which deviates largely from the norm, even though they do not necessarily outperform the less experienced maintainers on average. This

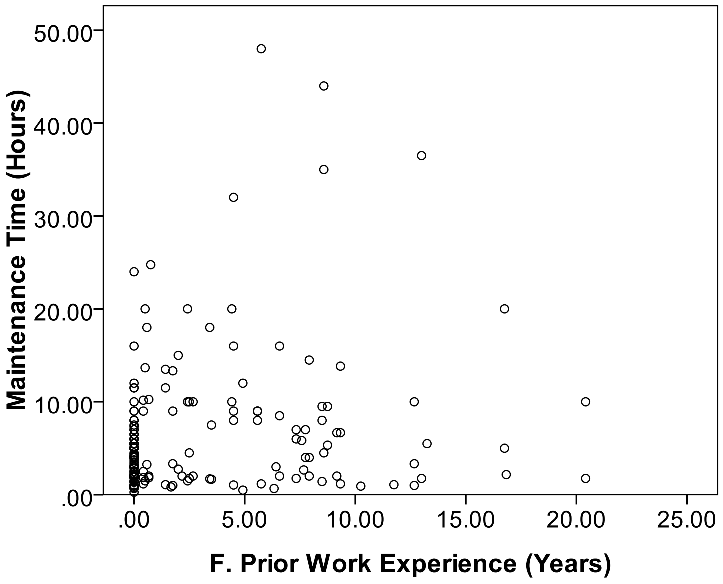


Figure 12. Scatterplot of maintenance time versus prior work experience (Factor F).

finding provides a fresh insight on the effect of prior work experience on software maintenance.

If the above observed phenomena are generally true in practice, then the implication might be as follows. While it may not be a bad idea to assign a maintenance task to a junior maintainer in the sense that the time required may, on average, be comparable to that by a senior maintainer, it may still be wiser to assign the task to the latter, as chances are that they will be able to complete the task correctly on time.

4.2.2. Case Studies

To obtain a better understanding of any characteristics of the program that might influence the subject to utilize the deployed design patterns to complete a change task or not, we examined some sample submissions to conduct qualitative analysis in the form of case studies. We first explain how we selected the sample submissions for analysis.

For each change task, we identified the set of all programs that were regarded as a correct program, that is, passed all our functional tests. Then, for each of these correct programs, we tracked the change of source code made from the given *JHotDraw* version to the submitted version. Based on the tracked changes, we constructed a class modification list of each program, that is, the set of all existing classes (from the given *JHotDraw* version), which had been either modified or inherited from new classes. We collected all those programs having the same modification list into the same group. For each group, we arbitrarily selected one program for investigation. Altogether, we found two groups per change task, as depicted in Figures 14–16. Here, programs in the same group were implemented by using the same solution strategy, but the resulting solutions were not necessarily the same.

Figure 14 depicts the relevant classes in *JHotDraw* at various stages for the *Audit* change task: before deployment of design patterns to support the task (top left diagram), after deployment of these design patterns (top right diagram), a pattern-aware solution that utilize these design patterns (bottom right diagram), and a pattern-unaware solution that does not utilize these design patterns (bottom left diagram). Moreover, the thick (blue) arrows indicate that the subjects either implement a pattern-unaware solution from the original version with (or as if there were) no design patterns deployed to support the change task, or a pattern-aware solution when given a refactored version with design patterns deployed to support the change task. In the figure, all classes added for deployment of design patterns to support the change task are shaded in dark, whereas the classes modified by the subjects to complete the change are shaded in golden yellow.

Similarly, Figure 15 and Figure 16 depict the corresponding relevant classes for the *Image* and *Language* change tasks, respectively. In each of these two figures, there is additionally a thick (blue) arrow pointing from the top right diagram to the bottom left diagram, indicating that some subjects chose to implement a pattern-unaware solution even though they were given a refactored version with design patterns deployed to support the change task.

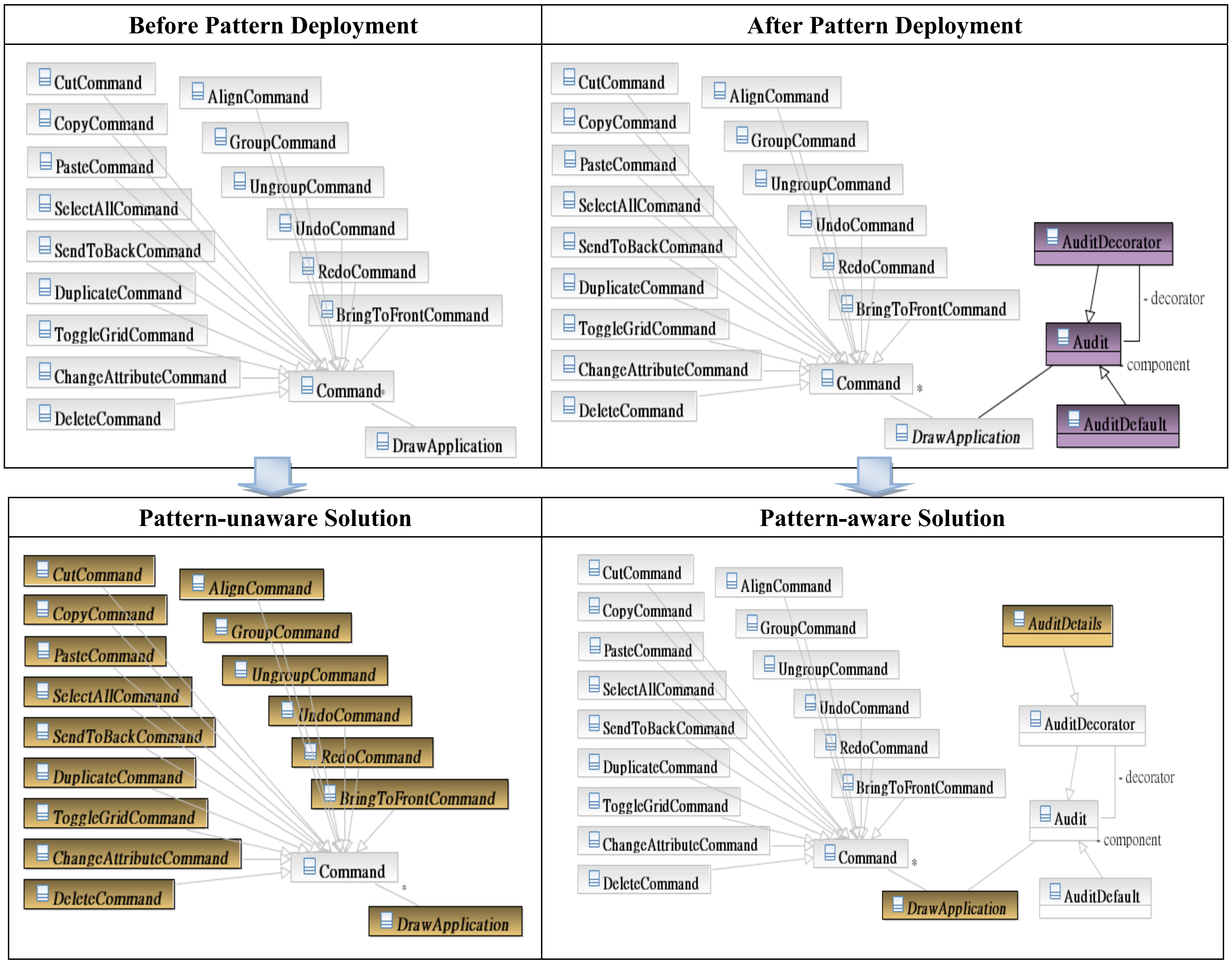
Audit. Figure 14 shows four diagrams that illustrate different stages of software maintenance for the *Audit* task. The top left diagram shows that *Command* is an interface which has been implemented in 15 ways, represented by 15 subclasses in the program. The task requirement of *Audit* is to keep the command history. To complete the task, one way is to modify the setting at the business logic level, which means to modify the class hierarchy rooted by the *Command* class. This solution is shown in the bottom left diagram of Figure 14. However, this approach is naïve and not satisfactory as it requires repetitive additions of similar statements in the solution.

Apart from that the most direct (pattern-unaware) solution, at the business logic level, one possible solution is to add a new auxiliary class to consolidate the actual manipulation of the command history and modify each of the concrete subclasses of *Command* to divert all such manipulation to this newly added class. To ease our discussion, let us call this class *Aux*. This solution is less direct, but can reduce the amount of repetitive code. However, this solution still requires modifying the classes as shown in the bottom left diagram of Figure 14. In addition, this solution needs the provision of the new class *Aux* for an extra level of code modularity. From the submitted programs, we found that all those who attempted the above-mentioned solutions failed to produce correct solutions. This may indicate that even though the solutions are feasible, subjects had to modify too many classes and could easily miss updating some important classes within the given time constraint of the submission deadline

On the other hand, the study shows that code modularity does have an effect on the subject's choice of maintenance strategy. Let us consider the top right diagram of Figure 14. The subject program is now deployed with a pattern that is rooted by the *Audit* interface. Although, in our opinion, the amount of code to be written to implement the required subclasses of *Audit* to support the change task is similar to that of *Aux*, yet the majority of the subjects chose this solution. Intuitively, the change task is coupled with the nature of each individual "command" more strongly than with the class that draws the canvas. Thus, adding a design pattern closer to the *Command* hierarchy can be a natural choice. Nonetheless, no subject who correctly completed the change task chose to add such a pattern to the *Command* hierarchy or adopted the above *Aux* class approach. We conjecture that the amount of repetitive addition of similar statements was an important consideration for the subjects to utilize the deployed design patterns to complete a change task.

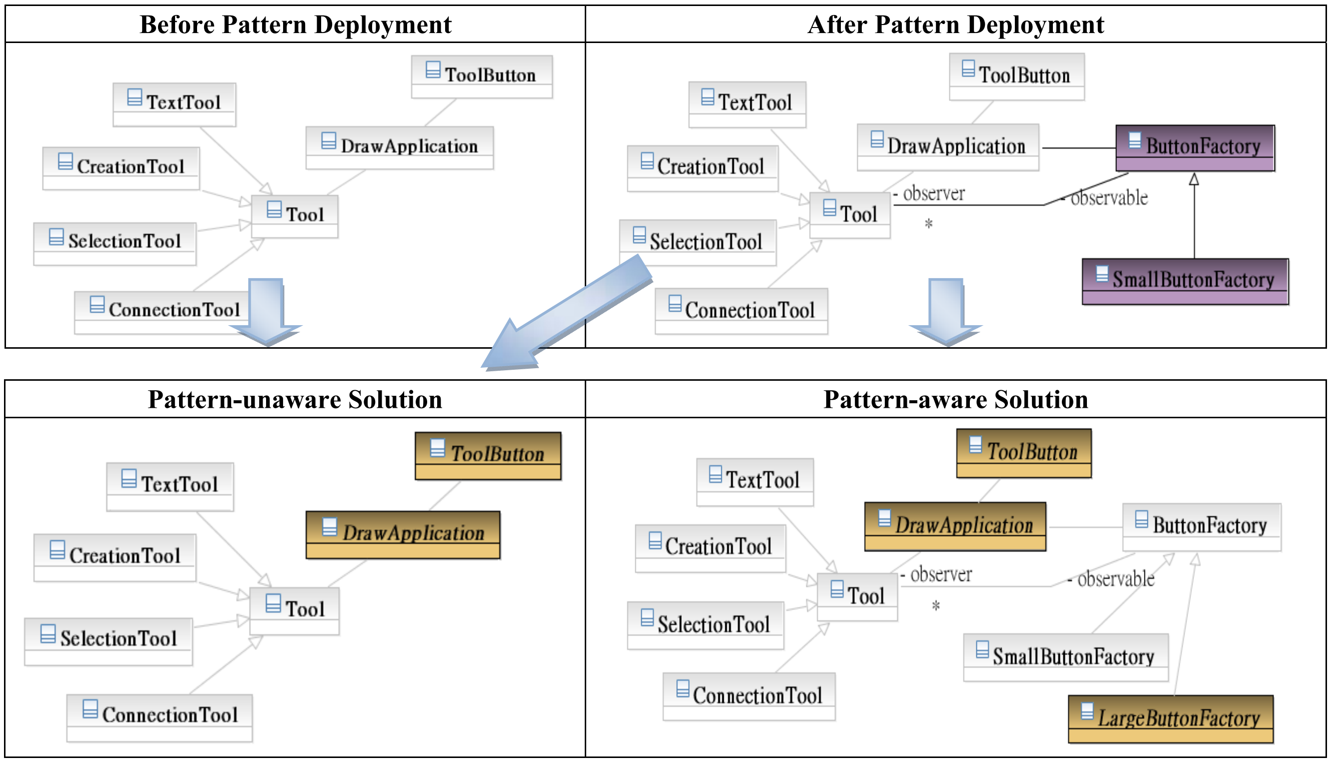
Image and Language. The modified classes to complete *Image* are depicted in Figure 15. Unlike the situation in *Audit*, more classes need to be modified in a pattern-aware solution than in a pattern-unaware solution for *Image*. Moreover, some subjects who worked on the refactored version still implemented pattern-unaware solutions to complete *Image*. We examined their submissions and found that some subjects indeed once considered a pattern-aware solution, but finally decided to implement pattern-unaware solutions. We conjecture that the decision is based on the simplicity of their final solutions. Still, some other subjects chose the pattern-aware solution and could complete the change task correctly.

Unlike *Audit*, which requires modifying 15 classes if not using the deployed design pattern, for *Image*, the submitted solutions and our own assessment show that at a minimum, only two classes (namely *DrawApplication* and *ToolButton*) may



Note: (1) Top-right: Classes added to *JHotDraw* for deploying design patterns to support *Audit* are shaded in purple.
(2) Bottom-left and bottom-right: Classes modified or added for completing the change task are shaded in golden yellow.

Figure 14. Modified classes in the refactored version and their pattern-aware or pattern-unaware solutions for *Audit*.



Note: (1) Top-right: Classes added to *JHotDraw* for deploying design patterns to support *Image* are shaded in purple.
(2) Bottom-left and bottom-right: Classes modified or added for completing the change task are shaded in golden yellow.

Figure 15. Modified classes in the refactored version and their pattern-aware or pattern-unaware solutions for *Image*.

require modification to implement the change task. Furthermore, there is no subclass of either class in the submitted solutions. The empirical result seems to indicate that if there is little replication of code required for a subject to add or modify, even when a strong hint of having a class known as *SmallButtonFactory* is presented to a subject, the subject would still have no motivation to use such deployed design patterns in their work.

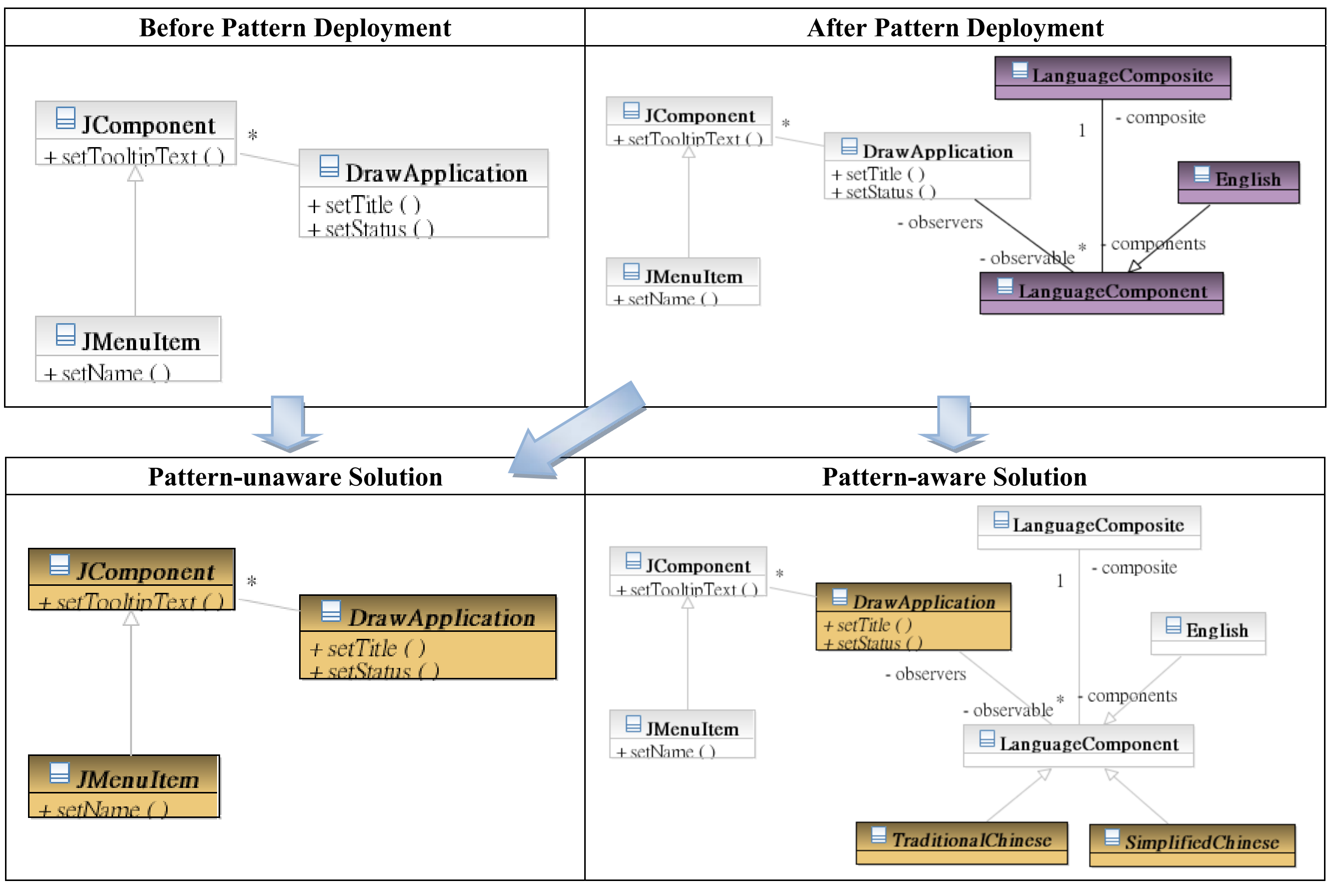
Figure 16 shows the classes modified by many subjects to complete *Language*. For both pattern-aware and pattern-unaware solutions, these subjects modified three classes. We find that many subjects did not commit to develop pattern-aware solutions. We also observe that the modified or newly added classes in their solutions typically do not entail any subclass. This finding is similar to what we have discussed for the *Image* task.

In summary, we find that from the maintainer's viewpoint, the amount of code replication seems to be a strong consideration on whether to utilize the deployed design patterns to implement a change task.

4.2.3. Statistical Analysis

We further conducted statistical tests to determine whether the factors are indeed correlated to the amount of maintenance time spent on the change tasks.

Data Sets and Outliers. As discussed at the beginning of Section 0, the whole data set under analysis consists of 164 correct program submissions. For ease of reference, we shall also name the whole data set as *Dataset W*. As a common practice, prior to performing the statistical analysis, we first visually examined the scatter of the data points, trying to identify any central pattern of associations. In doing so, we noted that 11 of the data points appear to reside clearly apart from the others. In statistics, a data point of this type is known as an *outlier* [40], which refers to an observation that lies outside the overall pattern of a distribution [30]. Outliers are often difficult to account for and yet may potentially skew the analysis results.



Note: (1) Top-right: Classes added to *JHotDraw* for deploying design patterns to support *Language* are shaded in purple.

(2) Bottom-left and bottom-right: Classes modified or added for completing the change task are shaded in golden yellow.

Figure 16. Modified classes in the refactored version and their pattern-aware or pattern-unaware solutions for *Language*.

There is no universal and rigid mathematical definition of what constitutes an outlier. Whether or not an observation is an outlier is ultimately a subjective exercise [30]. There are two mainstream approaches to the identification of outliers. The first is by visual examination, which we adopted in the statistical analysis as reported in a preliminary version [35] of this paper. The statistical analysis was based on a subset of the whole data set formed by excluding the aforementioned 11 data points that are visually apart from the rest. We shall refer to this subset of data as *Dataset Q*, which consists of 153 program submissions and was analyzed in the preliminary version [35] of this paper.

The second mainstream approach is to follow the "convenient definition" that a data point is an outlier if it "falls more than 1.5 times the interquartile range above the third quartile or below the first quartile" [40]. Obvious advantages of this definition are that it is both systematic and completely objective. In any case, we decided to consider this "convenient definition" for further analysis. By adopting this definition, we identified a different set of 11 data points as outliers. Excluding these from the whole data set results in *Dataset P* consisting of 153 program submissions. Dataset P is a different subset (of Dataset W) from Dataset Q, even though they, coincidentally, contain the same number of program submissions.

Finally, we performed additional statistical analysis based on the last two data sets (namely, Dataset W and Dataset P) to compare and triangulate with the results obtained by using only Dataset Q and previously analyzed in [35].

Correlation of Factors with Maintenance Time. We examine the correlation coefficient (CC), as well as its significance value, of each factor with the time taken to correctly complete a change task. Table 3 shows these values, which are obtained by using SPSS [42] to perform linear (multiple) regression analysis for each of the three data sets. The table also shows the rank of the factors in order of the correlation coefficients. We adopt the common criterion that a significance value below 0.05 is considered statistically significant. To facilitate reading, the significance values are shown with (✓) when it is statistically significant at 0.05 level, and with (×) otherwise.

From the analysis of both Dataset W and Dataset P, we find that only the factors of the first three ranks in the list are notably significant. They are, respectively, Factor A (Deployment of design patterns), Factor D (Prior exposure to the program), and Factor B (Presence of pattern-unaware solutions). Thus, there is statistical evidence that these three factors correlate significantly with the amount of maintenance time. The two lowest ranked factors, namely, Factor E (Prior exposure to the programming language) and Factor C (Prior exposure to design patterns), are insignificant.

The results obtained from Dataset Q and reported previously in [35] are slightly different from the above. While the first two ranked factors are the same (and both are statistically significant), as well as the last two ranked factors (both of which are statistically insignificant), the other two factors (Factor B and Factor F) happen to interchange in ranking. Interestingly, while Factor B is statistically significant in both the Dataset W and Dataset Q at the 0.05 level, it was not so in the analysis of Dataset P. Moreover, Factor F is statistically insignificant in the analysis of all data sets.

We further consider the change of significance values due to the possible effect of the outliers. Figure 17 depicts the change of significance values as the number of outliers cut increases from 0 (Dataset W) to 11 (Dataset P). While the significance values of the statistically insignificant factors (Factors C, E and F) do fluctuate somewhat, those of the three top-ranked factors (namely, Factors A, D and B) are relatively unaffected by the outliers in the sense that their curves almost overlap completely with the x-axis.

Overall, the analysis of Dataset W and Dataset P gives essentially the same results, which slightly differ from that of Dataset Q. Given that the former two data sets are more systematically and objectively formed, and that their analysis results agree well, we tend to have more confidence of their results. Moreover, excluding the outliers using the objective "convenient definition" does not significantly affect the analysis results. Therefore, our discussions in other sections of this paper have been based primarily on the observations on the whole data set.

4.3. Summary and Discussions

Based on the above statistics and observations, we summarize and discuss our results as follows.

1. Our results show that Factor A (Deployment of design patterns) correlates significantly with the amount of time taken to correctly complete a change task. Moreover, this factor is the most dominant one among all factors under study. The implication is that while there may exist other factors such as human factors that may influence a maintainer's performance, deploying design patterns that support a change task indeed has its own individual, significant benefits in shortening the maintenance time.

Table 3. Correlation of factors with maintenance time.

Factor	**(Whole) Dataset W (164 submissions)**			**Dataset P (153 submissions, outliers excluded)**			**Dataset Q (153 submissions, outliers excluded)**		
	CC♠	**Sig. Value (< 0.05 ?)**	**Rank**	**CC♠**	**Sig. Value (< 0.05 ?)**	**Rank**	**CC♠**	**Sig. Value (< 0.05 ?)**	**Rank**
A. Deployment of design patterns	0.537	<0.001(✓)	1	0.511	<0.001(✓)	1	0.541	<0.001(✓)	1
B. Presence of pattern-unaware solutions	0.229	<0.001 (✓)	3	0.217	<0.001 (✓)	3	0.110	0.083 (×)	4
C. Prior exposure to design patterns	0.020	0.633 (×)	6	0.004	0.942 (×)	6	0.016	0.689 (×)	6
D. Prior exposure to the program	0.321	<0.001 (✓)	2	0.328	<0.001 (✓)	2	0.311	0.017 (✓)	2
E. Prior exposure to the programming language	0.046	0.419 (×)	5	0.043	0.460 (×)	5	0.022	0.581 (×)	5
F. Prior work experience	0.100	0.078 (×)	4	0.048	0.414 (×)	4	0.105	0.082 (×)	3

♠ CC = Correlation Coefficient

2. In our context, Factor D (Prior exposure to the program) means that a maintainer has previously worked on the program to complete a certain change task. Thus, with such an experience, the maintainer would take a shorter time to complete another change task for the same program than if the maintainer had no such experience. Even if the new change tasks was unrelated to the previous one, having prior exposure to the program was found to be significantly useful. It is probably because a maintainer would have better understanding of the general structure of the design and code, such as which deployed design patterns are present and they may or may not support which change tasks. When a maintainer performs another change task, the maintainer may recall from their memory about the code and design structure.

3. Factor B (Presence of pattern-unaware solutions) indicates that alternative solutions to pattern-aware solutions matter. If a simpler alternative to the pattern-aware solution is obviously available, a maintainer will tend to implement such an alternative solution. Such a situation could significantly affect the performance of maintainers to complete a change task, because the maintainers may need to spend time on deciding which solution to implement.

4. Regarding Factor F (Prior work experience), we find that maintainers with more work experience tend not to spend unduly long maintenance time in correctly completing a change task. This result suggests that the usefulness of work experience is different from what is commonly perceived. While one may intuitively think that a maintainer with more work experience would perform better, we can only conclude from our result that he/she would have a smaller chance to perform badly. Thus, Factor F complements Factor A (Deployment of design patterns) in the sense that the consideration of work experience is to avoid employing "bad" maintainers, while the consideration of deploying design patterns is to improve the performance of maintainers being employed.

In addition to the above positive results, we find no statistical evidence that Factor C (Prior exposure to design patterns) or Factor E (Prior exposure to the programming language) might significantly affect the performance of maintainers. We discussed with those subjects who had learned design patterns or Java for a considerable length of time but still performed unsatisfactorily in the study. They generally expressed the same feeling that learning a concept and applying it to solve problems are very different. To some extent, this phenomenon may account for the insignificance of these two factors.

5. Threats to Validity

5.1. Construct Validity

Construct validity refers to the degree of whether our study is measuring what it is purported to measure.

One threat to validity is the dependent variable, which is the time reported by the subject. We have told the subjects in the tutorial session and stated clearly in the assignment system that the time reported would not affect their academic results. All the subjects are educated adults. We have no reason to doubt the reported time is not the time that they actually spent on the work. Moreover, our study was not conducted in a laboratory setting. Rather, the change tasks were part of their take-home assignments. Each subject was required to complete three change tasks, each of which could take a number of hours, as shown in Section 4 (Figure 6 and Figure 8). It is impractical to keep the subjects in a laboratory. Issues of personal privacy also disallow us from micro-collecting the time-related data on individuals.

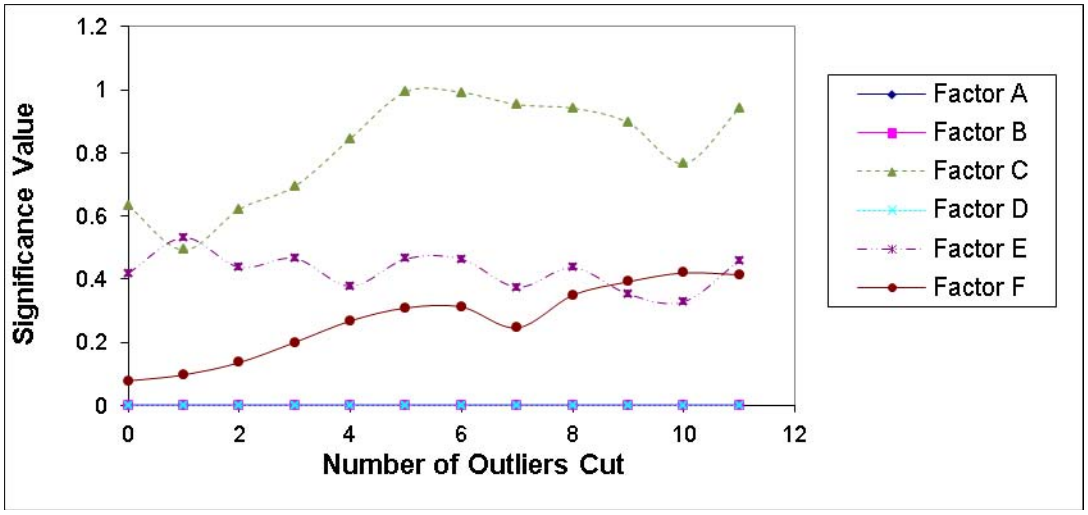


Figure 17. Change of significance values on correlation of factors with time taken as the number of outliers increase.

It appears that there is a large variation of the maintenance time spent, which can be accounted for by the diversity of human ability and, thus, we use statistical analysis to address this variation. We were also aware of the presence of an extreme data point (outlier) that may skew the statistics significantly. We have manually examined the data, ruled out the outlier from the reported analysis, and investigated the possible effect of other outliers in our analysis.

We have restricted the scope of our study to correctly completed maintenance tasks. As explained in Section 3.1, our context was limited by the course-based setting where incomplete changes might still be submitted by subjects. It was impossible for us to estimate how much time a subject may further need to complete the remaining part of his/her own submission. Because of this constraint, it would be a substantial threat for our study to draw conclusions from data generated from incorrect submissions.

5.2. Internal Validity

Internal validity concerns whether our findings truly represent a cause-and-effect relationship that follows logically from the design and operation of our study. Being aware of the potential experimental bias due to refactoring *JHotDraw* to deploy design patterns, we have arranged for independent verification of our refactoring process (before the subjects performed the change tasks) by five Hong Kong Ph.D. candidates in software engineering. They had a clear agreement on how and which particular design patterns should be deployed to support each change task.

Another potential source of interference is the possibility of undiscovered design patterns deployed in *JHotDraw*. If there were such undiscovered design patterns, the contribution of refactoring the original version using additional design patterns to support changes would be mixed with those undiscovered design patterns that also support the change tasks. To address this issue, we performed a code analysis on the versions of *JHotDraw* that we used in our study. In particular, we investigated each inheritance class hierarchy and ensured that our change tasks could not be completed by creating new subclasses of an existing hierarchy without replicating various existing codes in these new subclasses. It is because the replication of existing pieces of code in new subclasses generally violates the philosophy of using design patterns. We have also examined all the submitted program versions and found that no subjects utilized any undiscovered design patterns.

One may also worry that the design patterns deployed in the original version would have an impact on unrelated change tasks. For example, Version-A of *JHotDraw* is a variant of the original version with the addition of the *Decorator* pattern to support *Audit*. This version is supposed to be unrelated to *Image* and *Language*, but the added design pattern may distract the subjects when completing these unrelated change tasks. Although this distraction is possible, we do not believe it to be substantial because the introduced design patterns (for each of the refactored *JHotDraw*) have only added less than 100 lines of code to the original version, which has a total of 14,342 lines of code. Factors related to program comprehension may be interesting to warrant further study. The correctness and our coding styles may also have affected the productivity of subjects.

Another potential problem is the possibility of plagiarism by the subjects. We believe that the subjects in our study generally completed the changes by themselves, as we did not find high degrees of code similarities across the change parts of the submitted program versions using plagiarism checking tools provided by the assignment system used by the university. Subjects all submitted *JHotDraw* programs. It is impractical to identify plagiarism by directly comparing multiple *JHotDraw* versions. Finding the change part is not difficult as we used the "diff" utility to produce a change between a submitted version and the given version.

Another concern relates to the incentives of the subjects. Our subjects were students enrolled in software engineering courses rather than paid programmers. Their incentives were to finish the assignments before the deadline for partial fulfillment of the course requirements. One potential problem with this type of incentive is that they might not feel the urgency to complete the changes as soon as possible. Nevertheless, we found most students were active in seeking for clarifications of requirements and eager to achieve good grade in the course. We believe that the subjects had reasonably made their best effort in completing the assigned change tasks. To further address this issue, we plan to employ paid participants in future follow-up experiments.

In addition, students may have submitted incorrect versions because they ran out of time and could not finish their solutions. If more time (say, 6 weeks instead of 4 weeks) were given to the subjects, the number of correct submissions may have increased dramatically, and the results might not be the same. Regarding this issue, we find that the mean time spent from the correct and incorrect submissions were 7.13 days and 5.23 days, respectively (assuming 8 hours of work per day). Because the actual work time is much less than the allowed work time (that is, 4 weeks), we believe that the allowed work time is reasonable and extending it further would not have a significant effect on the number of correct submissions.

There may be other factors in addition to Factors A–F that may affect the dependent variable. In this experiment, we have made our best effort to consider both human and program factors. In comparison with related work, this study has already considered more factors than many others. Finally, in principle, it is possible that our survey on Factors C, E, and F did not capture the real data values. These data were, however, provided by the subjects. Although we had no way to verify the accuracy of the data, there is also no reason for us to doubt it either.

5.3. *External Validity*

The test of external validity questions the applicability and generality of our findings. The applicability of our findings must be carefully established. Only *JHotDraw* was used as the code base for changes in our study. Different results might be obtained from using different program versions with different requirements and nature. Also, in *JHotDraw*, not all kinds of design patterns were deployed. Different design patterns have characteristics that lead to distinctive pros and cons. More experiments are needed in this regard.

Another threat is the use of only three change tasks in our study. Using other types of change tasks might give rise to different results due to the distinct characteristics of the tasks. For example, a change can be adaptive, perfective or corrective. However, our study involves change tasks that require maintainers to investigate different system aspects, including control-flow, state transitions, event handling, and others. In addition, our selected change tasks are non-trivial. Thus, we do expect the selected change tasks to be somewhat similar to what maintainers might encounter in practice. Other orders of the change tasks may also affect the results. Similarly, using different change tasks on other *JHotDraw* versions or other applications are possible, yet the results may not be consistent with ours.

In our experimental design, there are only three class sections of subjects and their distribution is not even. This uneven distribution is due to practical constraints, as we cannot "design" the number of students in each class section. Regarding this, we examined the mean assignment scores of the subjects in different class sections after collecting their program submissions. For the *Audit* task, the mean scores of class sections 1, 2 and 3 are 93.54%, 79.02% and 77.78%, respectively. For the *Language* task, the respective mean scores are 73.87%, 79.95% and 84.41%. For the *Image* task, their respective mean scores are 76.92%, 83.23% and 94.20%. We do not find evidences of high-performance students or low-performance students having dominated any particular class section.

Eclipse and Java were used exclusively as the development tool and programming language, respectively. Doing so may limit the generality of the study to similar conditions. Nevertheless, Java is a popular object-oriented programming language, while *Eclipse* is a widely used tool in the industry to develop Java applications. Thus, the results should still be relevant.

In our experiment, we have included a tutorial session. Having the other kinds of sessions before a study may alter the results. Our subjects come from one university in Hong Kong. However, in this reported study, we did not guide or restrict our subjects to use or not to use any particular approach to develop their own solutions. Subject expertise composition may affect the way the subjects design their solution to complete a change task. Subjects having different cultural or family background, social or physiological status could also affect the results. Subjects from other universities may also have different behaviors.

One may also consider the change tasks as an independent variable. We agree with this viewpoint. However, different change tasks would naturally lead to different amounts of time spent. Even if a study concludes that the change tasks may produce comparable time spent, the results do not seem to be generalizable. Consequently, we did not analyze our results with the change tasks as an independent variable in our study.

6. Related Work

Gamma et al. [15] propose that design patterns be used to express object-oriented design experiences. They advocate that design patterns can reduce system complexity by referring abstractions as names with well-defined meaning. In Section 1, we have discussed related work that involves the study of human or program factors. We further discuss related work as follows.

One thread of research examines the quality of deployed design patterns. For instance, Vokáč [45] investigates whether classes participating in deployed design patterns are less fault-prone than those not participating in any deployed design patterns. We refer to the former kind of classes as participants and the latter kind as non-participants. Their study involves C++ programs and selects the *Decorator*, *Factory Method*, *Observer*, *Singleton*, and *Template Method* patterns. The results show that, on average, participants of the deployed *Observer* and *Singleton* patterns are more prone to faults than non-participants, whereas participants of the deployed *Factory Method* pattern are less prone to faults than non-participants. No statistically significant conclusion can be drawn for the *Decorator* and *Template Method* patterns.

Ng et al. [32] study the problem of quality of design patterns from another perspective. They conduct a controlled experiment, asking multiple groups of subjects to conduct the same set of perfective maintenance tasks on program versions deployed with design patterns that support the change tasks and on versions without such deployed design patterns but with the same functional behavior. They observe that using deployed design patterns does not necessarily produce less faulty code than not using design patterns to implement the changes.

Prechelt et al. [38] examine whether the documentation of deployed design patterns improves the functional quality of the maintainers to perform changes. Their results show that, with such documentation, the revised code has fewer faults than without documentation. The studies indicate that human aspects, such as the presence of and interpretation of documentation, should be considered in order that deployed design patterns can be useful.

One way to identify deployed design patterns in programs and, hence, obtain their documentations is to reverse-engineer patterns from their source code. Static code analysis tools like Ptidej [18] and Columbus [3] are available, but their accuracies vary [13]. De Lucia et al. [11][12] propose to first use a mining technique to find out the scope of individual patterns and then apply static code analysis. Their techniques can be applied to identify structural [11] and behavioral [12] design patterns. Tsantalis et al. [43] propose an approach to detect design patterns using similarity scoring between graph vertices. This approach has the ability to recognize the standard solutions as well as variants of design patterns.

Khomh and Guéhéneuc [25] find that design patterns in practice impact negatively on several quality attributes. They also identify some situations where the use of design patterns is not well justified. They warn practitioners that design patterns should be used with caution during development as design patterns may eventually hinder software maintenance and evolution.

More recently, Li et al. [28] have performed a case study to analyze the impact of the application of the *Bridge* design pattern to software safety using software fault tree analysis. The result indicates that deployed *Bridge* pattern could improve safety by two orders of magnitude because of the added redundancy to object-oriented design.

Another way to study the quality of deployed design patterns is to examine whether they can be used (in parts) to implement certain future requirements. Bieman et al. [7][8] perform two studies to examine whether class participants would have lower change counts than non-participants in the same programs when design patterns are deployed. In total, five (C++ and Java) software systems and twelve design patterns are examined. In only one of the systems are the participants changed less often than non-participants. In none of the other systems are any advantages of deployed design patterns shown over non-participants to be used to implement future requirements.

Aversano et al. [2] further reveal that the evolution of deployed design patterns depends on the purpose of their use in the examined software and does not quite depend on their types.

Di Penta et al. [10] examine whether different roles of participants may experience different evolution counts. They find that some roles receive more evolution counts than other roles, and changing the code of the participants of a role will likely change the classes coupled with such roles as well. A later study [26] shows that a participant of a deployed design pattern is also likely to be a participant of multiple deployed design patterns.

Ng et al. [32] show that it is more popular to modify concrete participants of deployed design patterns than other types of participants. Such findings support our study to examine the effect of design patterns developed with the aim to support target change requirements. Moreover, to single out the potential effect, the possible solutions of a change requirement should be capable of being implemented via adding or revising concrete participants of a deployed design pattern. Also, the deployed design patterns involved should not share participants with other deployed design patterns. Our study follows this setting.

As reported by Huston [21], different metrics may produce diverse and sometimes conflicting results on programs deployed with design patterns. Moreover, the checking of whether the deployed design patterns match their purpose cannot be evaluated directly through functional correctness [20]. Similarly to various related studies [37][38][39][46], our study adopts the time to complete a requirement as the metric.

The change tasks in our study involve changes of the user interface. Using deployed design patterns to develop user interfaces is not new. For instance, Ahmed and Ashraf [1] develop a model-based framework that uses design patterns as the interaction cores and thus lets developers add user interfaces around such patterns. Hennipman et al. [19] further propose a new category of pattern language specific to human-computer interaction. In our study, we use the traditional design patterns documented by Gamma et al. [16] rather than those specific design patterns tailored for user interface.

7. Conclusion

Software changes are inevitable. To facilitate maintainers to perform changes, deploying design patterns is attractive because prior design experience may be leveraged to resolve known design problems. However, other relevant factors may affect the performance of maintainers. Taking these factors into consideration, the fundamental question is whether the deployment of design patterns does facilitate software maintenance. In this paper, we have reported an experimental study to prioritize the relative importance of six factors in terms of the maintenance time needed by maintainers to complete a change task. In summary, we have the following observations.

1. Among the six factors, Factor A (Deployment of design patterns) is the most important factor. Thus, our answer to the fundamental question is affirmative. It suggests that this factor is not just a linear combination of other human and program factors, but rather has its individual contribution to fast maintenance. Also, this factor complements Factor F (Prior work experience) in the sense that the consideration of work experience is to avoid employing bad maintainers, while the consideration of deploying design patterns is to improve the performance of maintainers being employed.

2. Factor D (Prior exposure to the program) is of secondary importance, suggesting that when selecting maintainers, whether they have prior exposure to the program is notably more important than whether they have prior exposure to design patterns (Factor C) or the programming language (Factor E).

3. Factor B (Presence of pattern-unaware solutions) is a new factor identified as important in this paper compared with the related work. This factor suggests that if a simpler solution other than the pattern-aware solution is available, a

maintainer will tend to choose the simpler solution. As a guideline, developers should not simply deploy design patterns to support a change task just because the design patterns are applicable, but also need to consider whether there are alternative simpler solutions.

Further experiments can be conducted to revalidate our results in other environments or use our results as a basis toward further study. For example, it is interesting to know whether giving more time to the subjects contributes to receiving more submissions (producing software deliverables) of higher quality. Also, because we use the amount of time to correctly complete a change task as our metric, only correct program submissions were studied and analyzed. By properly modifying the design of the experiment, an investigation to the incorrect versions may produce additional insights. Future work may also investigate any interaction effects among the factors under study in this paper.

Acknowledgements

This paper is a substantially extended version of the preliminary one [35] presented at the 10th International Conference on Quality Software (QSIC 2010). We thank Zhifeng Lai, Heng Lu, Xinming Wang, Chang Xu, and Chunyang Ye for their kind help in validating the refactoring process in this study.

References

[1] S. Ahmed, and G. Ashraf, "Model-based User Interface Engineering with Design Patterns", *Journal of Systems and Software*, 80(8):1408–1422, 2007.

[2] L. Aversano, G. Canfora, L. Cerulo, C.D. Grosso, and M. Di Penta, "An Empirical Study on the Evolution of Design Patterns", in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (ESEC-FSE'07), ACM Press, Sep. 2007, pp. 385–394.

[3] Z. Balanyi, and R. Ferenc, "Mining Design Patterns from C++ Source Code", in *Proceedings of the 19th International Conference on Software Maintenance* (ICSM 2003), IEEE Computer Society Press, Sep. 2003, pp. 305–314.

[4] E.L.A. Baniassad, G.C. Murphy, and C. Schwanninger, "Design Pattern Rational Graphs: Linking Design to Source", in *Proceedings of the 25th International Conference on Software Engineering* (ICSE 2003), IEEE Computer Society Press, Mar. 2003, pp. 352–362.

[5] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley, 1997.

[6] K.H. Bennett, "Software Evolution: Past, Present and Future", *Information and Software Technology*, 39(11):673–680, 1996.

[7] J.M. Bieman, D. Jain, and H.J. Yang, "OO Design Patterns, Design Structure, and Program Changes: An Industrial Case Study", in *Proceedings of the International Conference on Software Maintenance* (ICSM 2001), IEEE Computer Society Press, Nov. 2001, pp. 580–589.

[8] J.M. Bieman, G. Straw, H. Wang, P.W. Munger, and R.T. Alexander, "Design Patterns and Change Proneness: An Examination of Five Evolving Systems", in *Proceedings of the 9th International Software Metrics Symposium* (METRIC 2003), IEEE Computer Society Press, Sep. 2003, pp. 40–49.

[9] CPPUnit, http://cppunit.sourceforge.net/ . (Last accessed: 16 Feb 2011)

[10] M. Di Penta, L. Cerulo, Y. Guéhéneuc, and G. Antoniol, "An Empirical Study of the Relationships between Design Pattern Roles and Class Change Proneness", in *Proceedings of the 24th IEEE International Conference on Software Maintenance* (ICSM 2008), IEEE Computer Society Press, Sep. 2008, pp. 217–226.

[11] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design Pattern Recovery through Visual Language Parsing and Source Code Analysis", *Journal of Systems and Software*, 82(7):1177–1193, 2009.

[12] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "Behavioral Pattern Identification through Visual Language Parsing and Code Instrumentation", in *Proceedings of European Conference on Software Maintenance and Reengineering* (CSMR 2009), IEEE Computer Society Press, Mar. 2009, pp. 99–108.

[13] A. De Lucia, E. Pompella, and S. Stefanucci, "Assessing the Maintenance Processes of a Software Organization: An Empirical Analysis of a Large Industrial Project", *Journal of Systems and Software*, 65(2): 87–103, 2003.

[14] L.J. Fulop, T. Gyovai, and R. Ferenc, "Evaluating C++ Design Pattern Miner Tools", in *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation* (SCAM'06), IEEE Computer Society Press, Sep. 2006, pp. 127–138.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Abstraction and Reuse of Object-Oriented Design", in *Proceedings of the 7th European Conference on Object-Oriented Programming* (ECOOP 1993), Springer-Verlag, July 1993, pp. 406–431.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[17] P. Gestwicki, and F.S. Sun, "Teaching Design Patterns Through Computer Game Development", *ACM Journal of Educational Resources in Computing* 8(1):1–22, 2008.

[18] Y.-G. Géhéneuc, and G. Antoniol, "DeMIMA: A Multi-layered Framework for Design Pattern Identification", *IEEE Transactions on Software Engineering*, 34(5):667–684, 2008.

[19] E. Hennipman, E. Oppelaar, and G. Veer, "Pattern Languages as Tool for Discount Usability Engineering", in *Interactive Systems. Design, Specification, and Verification: 15th International Workshop*, Springer-Verlag, July 2008, pp. 108–120.

[20] N.L. Hsueh, P.H. Chu, and W. Chu, "A Quantitative Approach for Evaluating the Quality of Design Patterns", *Journal of Systems and Software*, 81(8):1430–1439, 2008.

[21] B. Huston, "The Effects of Design Pattern Deployment on Metric Scores", *Journal of Systems and Software*, 58(3):261–269, 2001.

[22] JavaBeans with Observers, http://java.sun.com/developer/JDCTechTips/2006/tt0113.html#2 . (Last accessed: 16 Feb 2011)

[23] JHotDraw, http://sourceforge.net/projects/jhotdraw/ . (Last accessed: 16 Feb 2011)

[24] J. Kerievsky, *Refactoring to Patterns*. Addison-Wesley, 2005.

[25] F. Khomh, and Y.-G. Guéhéneuc, "Do Design Patterns Impact Software Quality Positively?", In *Proceedings of the 12th Conference on Software Maintenance and Reengineering* (CSMR 2008), IEEE Computer Society Press, Apr. 2008, pp. 274–278.

[26] F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "Playing Roles in Design Patterns: An Empirical Descriptive and Analytic Study", in *Proceedings of the 25th IEEE International Conference on Software Maintenance* (ICSM 2009), IEEE Computer Society Press, Sep. 2009, pp. 83–92.

[27] K.G. Kouskouras, A. Chatzigeorgiou, and G. Stephanides, "Facilitating Software Extension with Design Patterns and Aspect-Oriented Programming", *Journal of Systems and Software*, 81(10):1725–1737, 2008.

[28] G. Li, M. Lu, and B. Liu, "SFTA Based Safety Analysis for Bridge Pattern", in *Proceedings of the 8th International Conference on Reliability, Maintainability and Safety* (ICRMS 2009), IEEE Computer Society Press, Sep. 2009, pp. 522–525.

[29] B.P. Lientz, E.B. Swanson, and G.E. Tompkins, "Characteristics of Application Software Maintenance", *Communications of the ACM*, 21(6):466–471, 1978.

[30] D.S. Moore, and G.P. McCabe, *Introduction to the Practice of Statistics*. 3rd edition, New York: W.H. Freeman, 1999.

[31] T.H. Ng and S.C. Cheung, "Enhancing Class Commutability in the Deployment of Design Patterns", *Information and Software Technology*, 47(12):797–804, 2005.

[32] T.H. Ng, S.C. Cheung, W.K. Chan, and Y.T. Yu, "Do Maintainers Utilize Deployed Design Patterns Effectively?", in *Proceedings of the 29th International Conference on Software Engineering* (ICSE 2007), IEEE Computer Society Press, May 2007, pp. 168–177.

[33] T.H. Ng, S.C. Cheung, W.K. Chan, and Y.T. Yu, "Toward Effective Deployment of Design Patterns for Software Extension: A Case Study", in *Proceedings of the 4th Workshop on Software Quality* (WoSQ 2006)*, in conjunction with the 28th International Conference on Software Engineering* (ICSE 2006), ACM Press, May 2006, pp. 51–56.

[34] T.H. Ng, S.C. Cheung, W.K. Chan, and Y.T. Yu, "Work Experience versus Refactoring to Design Patterns: A Controlled Experiment", in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (SIGSOFT'06/FSE–14), ACM Press, Nov. 2006, pp. 12–22.

[35] T.H. Ng, Y.T. Yu, and S.C. Cheung, "Factors for Effective Use of Deployed Design Patterns", in *Proceedings of the 10th International Conference on Quality Software* (QSIC 2010), IEEE Computer Society Press, July 2010.

[36] Microsoft COM, http://www.microsoft.com/com/ . (Last accessed: 16 Feb 2011)

[37] G. C. Porras, and Y.-G. Guéhéneuc, "An Empirical Study on the Efficiency of Different Design Pattern Representations in UML Class Diagrams", *Empirical Software Engineering*, 15(5):493–522, 2010.

[38] L. Prechelt, B. Unger, M. Philippsen, and W.F. Tichy, "Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance", *IEEE Transactions on Software Engineering*, 28(6):595−606, 2002.

[39] L. Prechelt, B. Unger, W.F. Tichy, P. Brössler, and L.G. Votta, "A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions", *IEEE Transactions on Software Engineering*, 27(12):1134−1144, 2001.

[40] J. Renze, "Outlier.", from *MathWorld — A Wolfram Web Resource*, created by Eric W. Weisstein, http://mathworld.wolfram.com/Outlier.html . (Last accessed: 16 Feb 2011)

[41] M.P. Robillard, W. Coelho, and G.C. Murphy, "How Effective Developers Investigate Source Code: An Exploratory Study", *IEEE Transactions on Software Engineering,* 30(12):889−903, 2004.

[42] SPSS, http://www.spss.com/ . (Last accessed: 16 Feb 2011)

[43] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S.T. Halkidis, "Design Pattern Detection Using Similarity Scoring", *IEEE Transactions on Software Engineering*, 32(11):896−909, 2006.

[44] Tutorial Website, available at http://www.cs.cityu.edu.hk/~cssam/IST2011/workshop.html .

[45] M. Vokáč, "Defect Frequency and Design Patterns: An Empirical Study of Industrial Code", *IEEE Transactions on Software Engineering*, 30(12):904−917, 2004.

[46] M. Vokáč, W. Tichy, D.I.K. Sjøberg, E. Arisholm, and M. Aldrin, "A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns: A Replication in a Real Programming Environment", *Empirical Software Engineering* 9(3):149−195, 2004.

[47] C. Wohlin, P. Runeson, M. Höst, M. Ohlson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.