Do Maintainers Utilize Deployed Design Patterns Effectively? *

T.H. Ng[†]
City University of Hong Kong
cssam@cs.cityu.edu.hk

S.C. Cheung[‡]
Hong Kong University of Science and Technology
scc@cse.ust.hk

W.K. Chan
City University of Hong Kong
wkchan@cs.cityu.edu.hk

Y.T. Yu
City University of Hong Kong
csytyu@cityu.edu.hk

Abstract

One claimed benefit of deploying design patterns is facilitating maintainers to perform anticipated changes. However, it is not at all obvious that the relevant design patterns deployed in software will invariably be utilized for the changes. Moreover, we observe that many well-known design patterns consist of three types of programming elements (called participants), and that performing an anticipated change typically entails multiple tasks related to different types of participants. This paper studies empirically whether maintainers utilize deployed design patterns, and when they do, which tasks they more commonly perform. Our experiments show that almost all subjects perform the task of adding new concrete participants, fewer perform the tasks involving clients, whereas even fewer perform the tasks involving abstract participants. Furthermore, utilizing deployed design patterns (by performing whichever of the corresponding tasks) is found to be statistically associated with the delivery of less faulty codes.

Keywords: Empirical study, software maintenance, design patterns, faulty codes

1. Introduction

Changes in software are frequent throughout its development process. To keep the software useful, changes have to be continuously made even after it has been put into production use [8]. Software changes, however, have to be properly managed so that the code quality will not deteriorate rapidly to affect the software's usefulness [3]. One common means to facilitating anticipated changes during maintenance is to deploy design patterns in software at the time of development. Examples are the two well-known open source tools, *Eclipse* [4] and *JUnit* [7], in which the *design patterns* collected by Gamma *et al.* [5] were intentionally deployed and documented. Each design pattern packages a design solution that has been known as the best practices for solving a class of recurring problems [2]. When the need for an anticipated change arises, it is expected that the maintainers will effectively utilize the relevant design patterns deployed in the software to complete the change by performing specific tasks entailed in the patterns.

Nevertheless, when faced with a required change, maintainers must discover and implement a strategy to perform the change [13]. During this process, the maintainers may or may not be aware that relevant design patterns have been deployed in the software, even if they have learnt design patterns [11]. Moreover, even assuming that the maintainers know (say, from reading the documentation) that relevant design patterns have been deployed, they may or may not be able to identify the parts of code related to the patterns. In this regard, controlled experiments done by Prechelt *et al.* [11] have shown that explicit documentation of the design patterns would be very helpful. Furthermore, even so, the maintainers may consider that another alternative, which directly modifies the code without utilizing the properties of the deployed design patterns, is a simpler or more desirable solution for the change.

Thus, it is not at all obvious that the design patterns deployed in the software will invariably be utilized by

* The work described in this paper was partially supported by a grant (project no. HKUST DAG05/06.EG40) from Hong Kong University of Science and Technology, and a grant (project no. CityU 1166/04E) from the Research Grants Council of the Hong Kong Special Administrative Region, China.

† T.H. Ng is a part-time PhD student in the Department of Computer Science and Engineering, Hong Kong University of Science and Technology.

‡ All corresponding should be addressed to S.C. Cheung at the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong. Tel: +852-2358-7016. Fax: +852-2358-1477. Email: scc@cse.ust.hk

the maintainers to complete an anticipated change. We therefore first pursued the following research question:

> *Given a software system with relevant design patterns deployed and documented, how likely will its maintainer utilize the design patterns to complete an anticipated change?*

This question is important because while design patterns may be beneficial in providing higher flexibility, their deployment does incur non-negligible costs. Simpler solutions may exist when designing and implementing the software without the use of patterns [12]. If the design patterns turn out to be rarely utilized by the maintainers, why should these patterns be deployed in the first place?

We furthermore observe that many well-known design patterns consist of three types of programming elements (called participants), namely, *concrete participants*, *abstract participants* and *clients*, and that performing an anticipated change typically entails the completion of one or more tasks (referred to in this paper as *T1*, *T2* and *T3*, respectively) related to different types of participants. Details of the types of participants and tasks involved will be described in Section 2 of this paper.

For example, the anticipated change of adding a new locale setting in a hotel management system may involve the addition of a new concrete subclass (task *T1*) of the *Locale* class, or perhaps additionally the insertion of a new method to the *Locale* class hierarchy (task *T2*). As such, we are also interested in the following refined research questions:

(1) Given a software system with relevant design patterns deployed and documented, how likely will its maintainers perform each of the three tasks (T1, T2 and T3) to complete a given anticipated change?

(2) Does the likelihood of maintainers performing the three tasks differ significantly? In other words, do maintainers have a higher tendency to perform one task than another?

These research questions are also important because their answers will deepen our understanding of *how*, not just *whether*, a maintainer tends to utilize deployed design patterns. Finally, we also seek to address the issue of whether the deployment of design patterns is utilized effectively to produce better quality (less faulty) code. We phrase our last research question as follows:

> *When completing an anticipated change, is the code produced by maintainers who utilize the deployed relevant design patterns (by performing the T1, T2 and T3, respectively) less faulty than the code produced otherwise?*

We investigate these research questions experimentally, involving a total of 215 human subjects to perform 6 anticipated changes in 3 programs with a total of 17.8 KLOC and over 230 classes in 12 packages. Our experiments involve six design patterns that together cover all the three types of creational, structural and behavioral patterns proposed by Gamma *et al.* [11]. Section 3 of this paper will describe the detailed experiment setup and procedure.

Our experimental results show that almost all subjects perform the task (*T1*) of adding new concrete participants, a majority of the subjects perform the task (*T3*) involving clients, but on average only about half of the subjects perform the task (*T2*) involving abstract participants. This is interesting, indicating that the tasks do differ in popularity when a maintainer completes an anticipated change. We further found that the codes done by the subjects who utilize the deployed design patterns (by performing tasks *T1*, *T2* and *T3*, respectively) are significantly less faulty than the codes done by those who do not utilize the deployed design patterns. Section 4 of this paper will present the statistical results, and Section 5 will discuss the results and limitations of our experiments.

Our study is novel in two main aspects. First, to our knowledge, none of the previous empirical studies on design patterns investigated the question of whether maintainers utilize design patterns that are deployed in the software when they complete an anticipated change. Second, previous empirical studies on design patterns only treat an anticipated change as a single piece of work, whereas our study refines the change task into three (sub)tasks that involve different types of participants of the design patterns, and observes different popularity of utilization by the maintainers for each (sub)task.

Earlier, we reported an experiment to study whether the use of program refactoring to introduce additional design patterns is beneficial regardless of the work experience of maintainers [10]. Our earlier experiment did not consider the task breakdown involved in completing the anticipated changes. A more detailed review of related work will be presented in Section 6 of this paper. Finally, Section 7 concludes this paper.

2. Design patterns and their utilization

This section presents a review of design patterns and the types of the participants in a design pattern and the tasks involved in performing an anticipated change.

2.1. Design patterns

One of the most well-known sets of design patterns for object-oriented programs are the 23 design patterns collected by Gamma *et al.* [5]. For example, Figures 1 and 2 depict the *State* pattern and the *Visitor* pattern in UML notation, respectively. The former facilitates an object to alter its behavior by switching its context from one concrete state object to another. The latter extends the existing functions provided by an element without changing its contents. To describe the collaborations of the involved entities, these entities are termed as *participants* in the pattern category [5]. For instance, in Figure 1, the *State* pattern has three participants, namely, *Context*, *AbstractState* and *ConcreteState*.

2.2. Participants in design patterns

We have examined all 23 design patterns in [5] and found that the participants in every design pattern can be classified into the following three types[§]:

(a) *Concrete participant*: A participant which is a subclass of another participant. Examples are *ConcreteState*, *ConcreteVisitor* and *ConcreteElement* in Figures 1-2.

(b) *Abstract participant*: A participant which is a superclass of another participant. Examples are *AbstractState*, *Visitor* and *Element* in Figures 1-2.

(c) *Client participant*, or simply *client*: A participant which is neither a subclass nor a superclass of any participant, and it typically binds statically to an abstract participant. Examples are *Context*, *Client* and *ObjectStructure* in Figures 1-2.

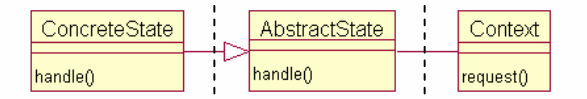

Figure 1. Structure of the *State* pattern [5]

[§] Note that the three types of participants may not be mutually exclusive.

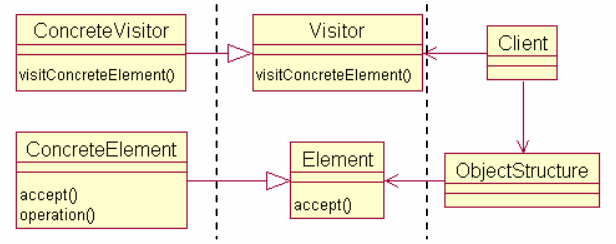

Figure 2. Structure of the *Visitor* pattern [5]

2.3. Tasks for a pre-deployed design pattern

In this section, we describe the kinds of tasks that are directly relevant to design patterns when completing an anticipated change [1]. We illustrate our explanation using a simplified version of a hotel management system. Since we are interested in the maintenance work related to design patterns, we consider only the tasks that will not destroy the existing pattern deployment. As such, we restrict our present study to perfective maintenance that adds new capabilities to an existing program.

To realize an anticipated change that is facilitated by a deployed design pattern, a maintainer may add new concrete participants, modify the existing interfaces of existing participants, or introduce new clients. For instance, recognizing the need to support different locale settings, maintainers could deploy the *State* pattern in the program to ease the anticipated changes of locale settings as shown in Figure 3. This deployment decouples a graphical user interface (GUI) from the text translation to the proper language.

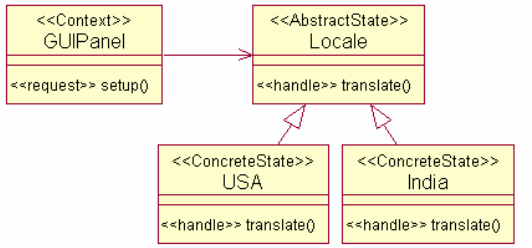

Figure 3. Deployment of the *State* pattern

Let us describe the three kinds of maintenance tasks which we have identified and abbreviated as *T1*, *T2* and *T3*, respectively.

T1. *Adding a new class as a concrete participant.* This task is to add a new class to join the pattern deployment as a concrete participant. For instance, if a new locale setting for Hong Kong has to be added to the above hotel management system, the maintainer would be expected to

perform the task of adding a new class that implements the locale setting to inherit the *Locale* class. Figure 4 depicts the resultant classes that participate in the pattern deployment after performing *T1*.

T2. *Modifying the existing interfaces of a participant.* The task is to add a new method to an existing interface of a participant for a pattern deployment. Typically, if the participant *P* is an abstract participant, it will also trigger the addition of a new method for the concrete participants of the abstract participant *P*. For example, if a change of the hotel management system is needed to display the currently selected country in the user interface, the maintainer would be expected to perform the task of adding a new method to the *Locale* class hierarchy returning an icon showing the corresponding information for a specific country. Figure 5 depicts the resultant classes that participate in the pattern deployment after performing *T2*.

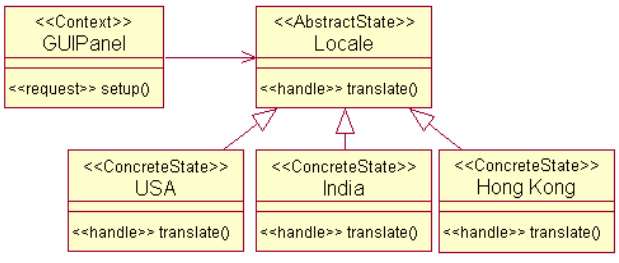

Figure 4. Deployment of the *State* pattern after performing *T1*

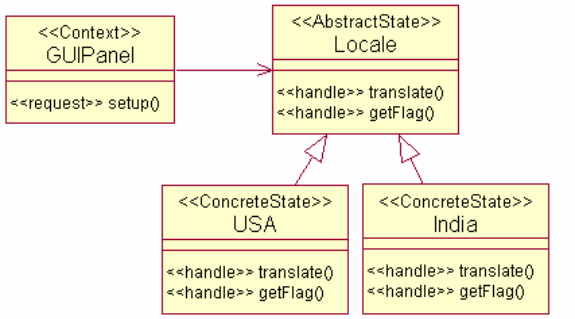

Figure 5. Deployment of the *State* pattern after performing *T2*

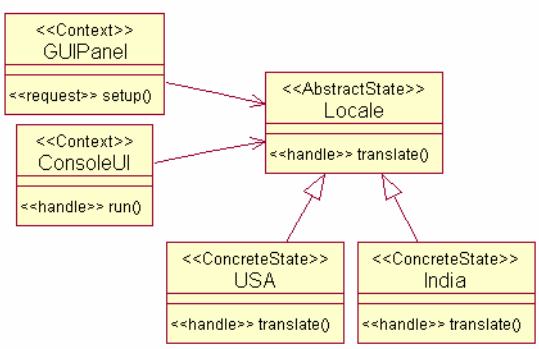

Figure 6. Deployment of the *State* pattern after performing *T3*

T3. *Introducing a new client.* This task involves the modification of an existing entity to invoke the design patterns. For example, if a change of the hotel management system is needed to support text translation in the console-based user interface that is previously co-supported with the GUI, the existing codes for that console-based user interface are expected to be modified to introduce new client codes for the *Locale* inheritance hierarchy. Figure 6 depicts the resultant classes that participate in the pattern deployment after performing *T3*.

There might be other tasks that do not involve any participant of the design pattern. Such tasks are considered to be not directly relevant to the design pattern and hence omitted in this study. For example, adding a Java statement to import new classes, such as "import javax.servlet.*;", is irrelevant to a change facilitated by design patterns. Moreover, the three tasks (*T1*, *T2* and *T3*) we introduce above may not form a complete list, but they are sufficient to frame our current study, as no other tasks relevant to the design patterns are involved in the anticipated changes considered in our experiments.

3. Methodology

This section presents the context and operational details of our empirical study.

3.1. Research questions

We are interested in examining the tendency of maintainers to use the deployed patterns to implement an anticipated change. As discussed in Section 2.3, we have identified three types of tasks (*T1*, *T2* and *T3*). We would like to know whether maintainers have

Table 1. Characteristics of the target programs used

Target program	Descriptive metrics	Deployed design patterns under study	Anticipated changes facilitated by the deployed design patterns under study
JHotDraw	15815 LOC over 211 classes in 10 packages	*FactoryMethod*, *Observer*	Support of new image icons displayed on the first four buttons on the tool bar.
		Composite, *Observer*	Support of new languages of the texts appearing in the menu bar.
MCM	1455 LOC over 15 classes in 1 package	*Command*	Support of new commands other than insertion, deletion and modification of individual appointments.
HMS	583 LOC over 10 classes in 1 package	*State*	Support of new operations other than check-ins and checkouts.

greater tendencies of performing one task than another. Moreover, we would like to know whether maintainers implement the change effectively. Hence, we also examine whether maintainers who perform each task deliver codes with fewer faults than those who do not perform the task. We therefore formulate the following research questions, which are refined from the corresponding questions we posed in Section 1:

Q1. What proportion of maintainers utilize the relevant design patterns in completing the required anticipated change?

Q1a. What proportion of them perform *T1*?

Q1b. What proportion of them perform *T2*?

Q1c. What proportion of them perform *T3*?

Q1d. Are the above proportions significantly different?

Q2. Comparing the codes delivered by maintainers who utlize the relevant design patterns in completing the required anticipated change to those delivered by maintainers who do not utilize patterns, are they equally faulty?

Q2a. Comparing those who perform *T1* with those who do not, are their codes equally faulty?

Q2b. Comparing those who perform *T2* with those who do not, are their codes equally faulty?

Q2c. Comparing those who perform *T3* with those who do not, are their codes equally faulty?

Furthermore, we formulate three two-tailed null hypotheses, hoping to reject them to establish the corresponding alternative hypotheses. In particular, the first hypothesis addresses question Q1d, and the remaining two hypotheses address questions Q2a-Q2c.

- **Null hypothesis (HA_0).** There is no significant difference, in terms of maintainers' tendency, in performing tasks *T1*, *T2* and *T3*.
- **Null hypothesis (HB_0).** There is no significant difference in the number of faults between the codes delivered by maintainers who perform and those who do not perform the tasks (*T1*, *T2* or *T3*) to complete the anticipated change.
- **Null hypothesis (HC_0).** There is no strong correlation between the number of faults in the delivered codes and whether or not a task (*T1*, *T2* or *T3*) is performed to complete the anticipated change.

3.2. Target programs

To achieve an acceptable level of generality, we studied target programs in several different application domains. The target programs used are as follows.

- *JHotDraw* – a drawing editor based on an open-source system [6] written in Java and with a graphical user interface (GUI).
- *MCM* – a multiple-user calendar manager supporting commands for manipulating individual appointments in a console-based user interface.
- *HMS* – a hotel management system prototype that supports the manipulation of records of hotel room usage. Through the system, a hotel receptionist may perform check-in and checkout operations for guests using a console-based user interface.

Table 1 summarizes the characteristics of each selected target program:

- **Descriptive metrics**. The target programs are in different application domains and with different sizes, ranging from a few hundreds to more than ten thousands of LOCs.
- **Deployed design patterns**. For each target program, we acted as an experienced maintainer to anticipate changes that are possibly required in the future. To prepare for these changes, design patterns have been deployed in the program in advance. Altogether, the design patterns relevant to this study include *Factory Method*, *Command*, *Composite*, *Observer* and *State*. *Factory Method* is a creational pattern, *Composite* is a structural pattern, and the other three are behavioral patterns. According to Gamma *et al.* [5], a design pattern can be creational, structural and behavioral in terms of

Table 2. Details of the changes required to be completed by the subjects

Target program	Required change	Description of the required change
JHotDraw	Image	A new menu "ToolBar Option" needs to be added. This menu would support a choice of "Small" or "Large" image. It will affect the size of the icons on all the buttons on the tool bar.
	Language	A new menu "Language" needs to be added. This menu would support a selection of "English", "Traditional Chinese", and "Simplified Chinese". When a new language, say English, is selected, the texts appearing in the menu bar, title bar, tool bar, and status bar need to be translated accordingly.
MCM	Group appointments	*MCM* needs to allow inserting, deleting and modifying group appointments by users in the same group in a GUI.
	Undo	All performed commands can be undone in a GUI setting.
HMS	GUI-based HMS	The user interface needs to be revised from console-based to GUI-based.
		Each command requires a separate GUI for receiving inputs and displaying outputs.
	Search	An enquiry operation is needed to support searches for occupants in a hotel.

its purpose. Thus, the selected design patterns cover all types of design patterns categorized in terms of their purposes.

3.3. Required changes

In general, a change can be adaptive, corrective, and perfective [9]. Table 2 presents the details of our required changes to the target programs. All these changes are perfective, which improve a system's functionality. Our experiment studied only perfective changes because various studies [1][9] have reported that perfective changes occur most frequently in software maintenance.

Table 3 summarizes the required changes associated with each task. We have selected the changes so that each task (*T1*, *T2* and *T3*) is applicable to at least two required changes.

We were also conscious of the reality of the changes. We have selected changes that were challenging enough to require maintainers to spend a non-negligible amount of effort investigating the program and completing the change. In fact, through a self-experimentation, we have ensured that each selected change requires us to spend at least three hours for completion, no matter whether it is done with or without using the deployed patterns.

Table 3. Tasks expected to be performed for completing the required changes

Task	Required changes
T1	*Image*, *Language*, *Group appointments*, *Undo*, *Search*
T2	*Image*, *Undo*, *GUI-based HMS*
T3	*Image*, *Language*

3.4. Subjects

Our experiments involved a total of 215 students who were enrolled in an undergraduate-level Java programming course offered by the Hong Kong University of Science and Technology. As we will discuss in Section 6, this size of subject set compares favorably with those involved in related empirical studies on design patterns, each of which employed not more than 118 subjects.

Our subjects were grouped to complete different changes. Group 1 consisted of 27 subjects performing *Image*, Group 2 consisted of 28 subjects performing *Language*, Group 3 consisted of 100 subjects performing *Group appointments* and *Undo*, and Group 4 consisted of 60 subjects performing *GUI-based HMS* and *Search*.

3.5. Experimental Procedure

To prepare the subjects to complete their required changes, a two-hour information session was devoted to education to review general object-oriented concepts and to explain the requirements of the changes. The materials presented to subjects were as follows:

- **Requirements Specification**. Subjects were required to achieve functional correctness for their assigned changes. In order to specify the functional requirements unambiguously, we presented to the subjects a demonstration program that implemented the required changes and deemed it as a functionally correct version. To avoid plagiarism, the program was given to them in the form of a binary executable in obfuscated Java bytecode, so that a decompiled version of the binary executables

is incomprehensible. We explicitly specified in the requirements that a submission of the decompiled version of the obfuscated code was prohibited and should automatically lead to no marks. Our analysis of the decompiled version shows that the effort needed to understand the decompiled version is greater than that needed to modify the given version to complete their assigned changes.

- **Source Code**. The source code of the program to be modified was presented to the subjects. The source code was in a form that can be compiled and run successfully using the standard Java compiler and interpreter, respectively.
- **Documentation**. The API documentation of the target program, which contains references of the design patterns deployed in the target program, was presented to the subjects. The UML models of each class hierarchy generated by reverse engineering tools were also presented to the subjects.

During the information session, we executed the demonstration program to describe the functional requirements. We distributed the API documentation and class hierarchies of the target program to the subjects without further elaboration. We gave the subjects freedom to determine their own strategies to complete the required changes.

Each subject was requested to carry out their assignment of software changes individually. The subjects were given one month to work on their assignments and they were allowed to submit their programs at any time before the deadline.

4. Results

In this section, we present our statistical findings of the empirical study. In summary, we find that the maintainers were more likely to perform the tasks (*T1*, *T2*, or *T3*) to implement a change that is applicable. Furthermore, they exhibited greater tendency to perform some tasks over other tasks. Specifically, we find that the "inserting a concrete participant" tactic is more often used than the "inserting a client" tactic, which is in turn more often used than the "modifying the existing interface of existing participants" tactic. We also find that maintainers who performed the tasks to implement a change did commit fewer faults than those do not. The details of our results are as follows.

4.1. Do maintainers utilize deployed design patterns?

Tables 4-6 present the distributions of the programs done by performing tasks *T1*, *T2*, and *T3*, respectively. We note that when the maintainers were given the chances to perform tasks *T1*, *T2*, and *T3*, respectively, to complete a change, most of them (96.73%, 57.54% and 75.59%, respectively) would utilize the relevant design patterns and perform the tasks. Overall, 79.13% of all cases performed such a task. This is an encouraging result to support the deployment of design patterns as they were utilized by most of the subjects to complete the anticipated changes.

However, the utilization percentages amongst the three tasks are far from even. Using the Kruskal-Wallis test, we studied if the three tasks are significantly different, in terms of whether a task is performed by a maintainer when it is applicable to a required change. From the test results (Table 7), we reject the hypothesis HA_0 and conclude that the three tasks differ significantly in popularity. On closer look, *T2* is the task with the smallest utilization percentage. In particular, only 39% of the maintainers use *T2* to complete the change GUI-based HMS. These statistics suggest that the nature of the task needed for utilizing the deployed design patterns has a significant effect on whether the maintainer will utilize the design patterns.

Table 4. Percentage of programs with T1 performed for the required changes

Required change	Number of programs	Number of programs with T1 performed	Percentage of programs with T1 performed
Image	22	22	100%
Language	23	23	100%
Group appointments	98	98	100%
Undo	98	82	83.67%
Search	51	51	100%
		Average	96.73%

Table 5. Percentage of programs with T2 performed for the required changes

Required change	Number of programs	Number of programs with T2 performed	Percentage of programs with T2 performed
Image	22	17	77.27%
Undo	98	55	56.12%
GUI-based HMS	51	20	39.22%
		Average	57.54%

Table 6. Percentage of programs with T3 performed for the required changes

Required change	Number of programs	Number of programs with T3 performed	Percentage of programs with T3 performed
Image	22	17	77.27%
Language	23	17	73.91%
		Average	75.59%

Table 7. Kruskal-Wallis test results on HA_0

Criterion	Chi-square result	Conclusion
HA_0	53.73	HA_0 is **rejected** at the 5% significance level ($H > 5.99$ when $df = 2$)

4.2. Do maintainers who utilize deployed design patterns deliver better codes?

Utilizing deployed design patterns does not necessarily mean that better codes are delivered. In this subsection, we analyze the performance of the subjects in terms of the functional correctness of their submitted programs, measured by the number of faults found.

Table 8 shows the empirical results (those change-task pairs with 100% design pattern utilization are omitted). There were 82 (out of 98), 92 (out of 169), and 34 (out of 45) maintainers who chose to perform *T1*, *T2* and *T3*, respectively. We observed from Table 8 that those programs for which the tasks were used to implement the changes were less faulty than those do not. Also, based on the Mann-Whitney test results as shown in Table 9, we reject the hypothesis HB_0.

We further calculated the Point-Biserial correlation, which is a measure of the strength of the relationship between whether a task is performed to implement the change and the number of faults. With this correlation, we performed the *t*-test for HC_0 and reject HC_0, as shown in Table 10. Thus, performing the tasks (*T1*, *T2* or *T3*) is statistically significantly associated with the delivery of less faulty codes.

Table 8. Number of faults found in programs

Required change	Task	Programs with the task performed?	Number of programs	Number of faults	
				Mean	**S.D.**
Undo	T1	Yes	82	2.63	2.75
		No	16	6.81	4.61
Image	T2	Yes	17	0.24	0.66
		No	5	5.2	0.45
Undo	T2	Yes	55	2.45	2.67
		No	43	4.42	4.04
GUI-based HMS	T2	Yes	20	1.29	1.52
		No	31	2.63	2.08
Image	T3	Yes	17	0.24	0.66
		No	5	5.2	0.45
Language	T3	Yes	17	0.76	0.97
		No	6	3.33	1.86

Table 9. Mann-Whitney test results on HB_0

Criterion	*z*	Conclusion
HB_0 with respect to T1 applicable to *Undo*	3.46	HB_0 is **rejected** at the 5% significance level ($\|z\| > 1.96$)
HB_0 with respect to T2 applicable to *Image*	3.2	
HB_0 with respect to T2 applicable to *Undo*	2.23	
HB_0 with respect to T2 applicable to *GUI-based HMS*	2.2	
HB_0 with respect to T3 applicable to *Image*	3.2	
HB_0 with respect to T3 to applicable to *Language*	2.59	

Table 10. Point-Biserial correlation coefficients and *t*-test results on HC_0

Criterion	r	t	Conclusion
HC_0 with respect to T1 applicable to *Undo*	-0.45	-4.91	HC_0 is **rejected** at the 5% significance level ($\|t\| > 1.99$ when $df = 96$)
HC_0 with respect to T2 applicable to *Image*	-0.99	-34.62	HC_0 is **rejected** at the 5% significance level ($\|t\| > 2.086$ when $df = 20$)
HC_0 with respect to T2 applicable to *Undo*	-0.28	-2.89	HC_0 is **rejected** at the 5% significance level ($\|t\| > 1.990$ when $df = 96$)
HC_0 with respect to T2 applicable to *GUI-based HMS*	-0.34	-2.65	HC_0 is **rejected** at the 5% significance level ($\|t\| > 2.010$ when $df = 49$)
HC_0 with respect to T3 applicable to *Image*	-0.99	-34.62	HC_0 is **rejected** at the 5% significance level ($\|t\| > 2.086$ when $df = 20$)
HC_0 with respect to T3 applicable to *Language*	-0.79	-5.90	HC_0 is **rejected** at the 5% significance level ($\|t\| > 2.080$ when $df = 21$)

5. Threats to validity

5.1. 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. A potential criticism on the internal validity is the plagiarism problem of the subjects. In our study, we found that the subjects generally asked questions actively during the information session. Before the deadline, the subjects kept asking for clarification by email. This increases our confidence that the subjects generally completed the tasks by themselves. Also, using plagiarism checking tools, we did not find high degrees of code similarity across submitted programs.

Another concern is that the subjects may have a wide variety of levels of expertise. In particular, the subjects who submitted codes with many faults may be due to poor programming skills rather than whether they utilized deployed design patterns. To address this issue, we also studied the performance of the subjects in their involved undergraduate courses. We found that the majority of these subjects obtained above-mean grades in the courses and their implementation strategies to complete their assigned tasks were indeed feasible approaches to achieve functional correctness.

5.2. External validity

External validity concerns the applicability and generality of our findings. Only *HMS*, *MCM* and *JHotDraw* were used as the target programs for changes. Different results might be obtained from using different programs with different requirements and nature. Also, in those programs, the design patterns deployed are not exhaustive. Different design patterns have characteristics that lead to distinctive pros and cons. However, the selected ones cover all types of design patterns in terms of their purpose (c.f. Section 3.2).

Another threat to the generality of our study is the use of only a finite number of perfective changes in our study. Although these changes are non-trivial ones requiring maintainers to investigate different system aspects (such as control-flow, state transitions, event handling), there exist many possible software modification types. For example, a change can be adaptive, perfective or corrective [8]. Clearly, each type has distinct characteristics. Still, our selected changes being so large that they cannot be completely understood in a short amount of time contributes to achieving an acceptable level of external validity.

5.3. Reliability

To facilitate replication of this experiment, all our experimental materials, including the source codes of the target and demonstration programs presented to the subjects for each study, are available to the public from http://www.cs.cityu.edu.hk/~cssam/ICSE/07.html

6. Related work

A number of empirical studies have previously been conducted on the modification of software by maintainers, but not all of them considered the effect of design patterns. For instance, Robillard *et al.* [13] researched on how effective maintainers investigated source code prior to performing changes to software, but did not specifically mention design patterns in their paper. Readers who are interested in studies within these contexts may refer to Robillard *et al.* [13] for an overview of the literature on the topic. In this brief survey, however, we shall focus on studies related to maintainers performing changes to software *in which design patterns had been deployed in advanc*e.

Prechelt *et al.* presented two reports of controlled experiments on the use of design patterns. In their first experiment [12], the same changes were performed on two groups of functionally equivalent programs: in one group of programs design patterns were deployed, while in the other group simpler alternatives but no design patterns were used. The results were found to depend on the particular design patterns being studied. While programs with the *Observer* pattern required more maintenance time than those without the pattern, programs with the *Decorator* pattern required less maintenance time than those without the pattern. The *Abstract Factory* and *Composite* patterns had no significant effect, and the results for the *Visitor* pattern were inconclusive.

In their second set of controlled experiments, Prechelt *et al.* [11] studied whether the explicit documentation of design patterns would be helpful to the maintainer. The results were positive in terms of both the time taken for the software changes and the functional correctness of the resulting software. *Composite*, *Observer*, *Template Method*, and *Visitor* were the design patterns investigated in the experiment.

The first experiment [12] of Prechelt *et al.* was subsequently repeated by Vokáč *et al.* [15] with paid industry professionals as subjects and done in a real programming environment so as to increase the experimental realism. However, the new results obtained were contrary to those of the original experiment. Vokáč *et al.* found that programs with the *Observer* pattern required *less* maintenance time than those without the pattern. They concluded that the question of whether the deployment of design patterns is beneficial cannot be generally answered, but depends on the nature of the problems and the deployed design

patterns. Vokáč's replicated study only considered the patterns *Observer*, *Decorator*, *Composite* and *Visitor*.

Earlier, we reported an experiment [10] to study whether the use of program refactoring to introduce additional design patterns is beneficial regardless of the work experience of maintainers. Our previous results showed that, to complete a maintenance task of perfective nature, the time spent by inexperienced maintainers on a refactored version was actually much shorter than the time spent by the experienced subjects on the original version (that is, the program version before refactoring). Moreover, the quality of the programs delivered by the two groups of subjects, in terms of correctness, was found to be comparable. The investigated design patterns were *Composite*, *Decorator*, *Factory Method* and *Observer*.

Several aspects clearly distinguish our work with the above-described studies. First, none of these studies consider our research questions of whether or not maintainers will, when completing anticipated changes, utilize the relevant design patterns that have been deployed and documented in the software at the time of development. Secondly, all previous empirical studies treat the utilization of deployed design patterns wholly as a single piece of work, whereas we have refined the work into three (sub)tasks that involve different types of programming elements called participants in the design pattern. Our experiment results indicate that maintainers seem to have a greater tendency to perform one kind of these tasks than another. This improves our understanding of how, not just whether, maintainers actually utilize deployed design patterns in completing anticipated changes.

In addition, we have further investigated whether the code quality delivered by maintainers who utilize the deployed design patterns is significantly better than that by maintainers who do not utilize patterns. With regard to this, the closest related work is the one reported in [14]. Vokáč [14] sought to find out any systematic correlation between the occurrence of certain design patterns and defect rates in the software. His analysis revealed that code related to some patterns (such as *Factory*) had a lower defect rate than the code in general, whereas some patterns (such as *Observer*) were correlated with higher defect rates. However, the defects he studied were not specific to those made during maintenance, nor did his work connect the defects to the tasks actually performed by maintainers when utilizing deployed design patterns. Indeed, our findings are more conclusive within our experimental context. Our finding is that regardless of the type of tasks performed by maintainers when utilizing deployed design patterns for anticipated changes, the delivered code is statistically significantly less faulty than the code done without utilizing patterns. This provides encouraging empirical evidence in support of the deployment of design patterns for the purpose of facilitating anticipated changes.

7. Conclusion

Existing empirical studies have validated that deploying design patterns increases program flexibility, despite the risk that it could complicate the program structure. However, if deployed design patterns are rarely utilized by maintainers, it would probably be not worthwhile to deploy them in advance. Thus, we conducted an empirical study to examine the potential gap between the following two scenarios: a) the deployment of design patterns in advance to prepare for anticipated changes, and b) the utilization of deployed design patterns to complete required changes that were anticipated at the time of deployment.

In our study, we refined an anticipated change facilitated by the deployment of design patterns into three finer-grained tasks, namely *T1*) adding new concrete participants, *T2*) modifying the existing interfaces of a participant, and *T3*) introducing a new client. We found that almost all subjects perform *T1*, fewer perform *T3*, whereas even fewer perform *T2*. Regardless of this, utilizing deployed design patterns by performing whichever of the finer tasks is found to be statistically significantly associated with the delivery of less faulty codes. Our results suggest that when *T1* is applicable to an anticipated change, there is no obvious gap between the deployment of design patterns and its utilization. However, when *T2* or *T3* is applicable to an anticipated change, further research is needed to raise the maintainers' awareness to perform these two tasks.

8. References

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

[2] K. Beck, R. Crocker, G. Meszaros, J. Vlissides, J.O. Coplien, L. Dominick, and F. Paulisch, "Industrial Experience with Design Patterns", in *Proceedings of the 18th International Conference on Software Engineering* (ICSE 1996), IEEE Computer Society Press, Berlin, Germany, Mar. 1996, pp. 103–114.

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

[4] Eclipse, http://www.eclipse.org/. (Last accessed: 5 Feb 2007)

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

[6] JHotDraw, http://sourceforge.net/projects/jhotdraw/. (Last accessed: 5 Feb 2007)

[7] JUnit, http://junit.sourceforge.net/. (Last accessed: 5 Feb 2007)

[8] M.M. Lehman, and L.A. Belady, *Program Evolution – Processes of Software Change*. Academic Press, 1985.

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

[10] 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, Portland, Oregon, USA, Nov. 2006, pp. 12-22.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.