Work Experience versus Refactoring to Design Patterns: A Controlled Experiment[*]

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
Program refactoring using design patterns is an attractive approach for facilitating anticipated changes. Its benefit depends on at least two factors, namely the effort involved in the refactoring and how effective it is. For example, the benefit would be small if too much effort is required to translate a program correctly into a refactorized form, and whether such a form could effectively guide maintainers to complete anticipated changes is unknown. A metric of effectiveness is the maintainers' performance, which can be affected by their work experience, in realizing the changes. Hence, an interesting question arises. Is program refactoring to introduce additional patterns beneficial regardless of the work experience of the maintainers? In this paper, we report a controlled experiment on maintaining *JHotDraw*, an open source system deployed with multiple patterns. We compared maintainers with and without work experience. Our empirical results show that, to complete a maintenance task of perfective nature, the time spent even by the inexperienced maintainers on a refactorized version is much shorter than that of the experienced subjects on the original version. Moreover, the quality of their delivered programs, in terms of correctness, is found to be comparable.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques – *object-oriented design methods*; D.3.3 [**Programming Languages**]: Language Constructs and Features – *patterns*

General Terms
Design, Experimentation, Languages

Keywords
Controlled experiment, refactoring, design patterns.

1. INTRODUCTION
To implement change requests, software maintainers should be able to comprehend the potential necessary changes and modify the program accordingly [5]. They may apply the notion of *design patterns* to facilitate anticipated changes. A popular catalog of design patterns is proposed by Gamma *et al.* [12], in which each records a design solution that has been developed and refined from prior design experience, solving a class of recurring problems in an object oriented approach. Design patterns encourage the best practices [3] and are widely used in the industry. For instance, JUnit [15] uses the *Composite* pattern, and J2EE EJB [13] and Microsoft COM [19] use the *Factory Method* pattern.

In software development, enhancing program flexibility for anticipated changes is an important driving force behind introducing design patterns [2]. A popular means to introduce design patterns in a program is via *program refactoring* [11]. To tackle different aspects of the anticipated changes, maintainers may deploy multiple design patterns in the program [12][26][27]. Systematic instructions [16] and tool support [22] have been proposed to assist and automate the refactorization process.

Refactoring is a *behavior-preserving transformation* [16]. Since its total automation is impractical, program refactoring is largely an error-prone manual process. In addition, whether such program refactoring would actually facilitate the anticipated changes is unknown. Intuitively, the performance of maintainers in realizing the changes could depend on their work experience. However, as we review in Section 2, the relationship between work experience and program refactoring has not been adequately studied. An interesting question thus arises. *Would refactoring a program using design patterns (in)conclusively supersede the effect of work experience to guide maintainers to complete a maintenance task, or vice versa?*

We empirically investigate the above question. In our controlled experiment, we replicated the realization of three maintenance tasks of perfective nature on two functionally equivalent programs, with and without design patterns to facilitate the required changes. We selected *JHotDraw* [14], an open-source medium-sized software program that had been deployed with multiple design patterns, as the testbed. For each task, we compared the following two approaches, involving those subjects (i.e., maintainers) with and without work experience.

- The subjects in the first approach performed the task directly on the original program.
- The subjects in the second approach performed the task on a refactored version of the original program using additional design patterns to facilitate the required changes.

[*] The work described by this paper was partially supported by grants (Project nos. CityU 1195/03E and HKUST 6187/02E) 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) 23587016. Fax: (+852) 23581477. Email: scc@cse.ust.hk.

[§] Parts of the work were done while W.K. Chan was with the Hong Kong University of Science and Technology before August 2006.

Our results show that when using the second approach, maintainers complete the change task faster than when using the first approach, regardless of their work experience. In addition, the quality of their delivered programs, in terms of the number of detected functional failures, is found to be comparable. The main contribution of this paper is to provide the first set of experimental results that support the use of design patterns in refactoring real-life software for anticipated changes.

The rest of this paper is organized as follows: Related work will be discussed in the next section, followed by an explanation of our methodology in Section 3. Then, we analyze the experimental results in Section 4, and discuss the factors affecting the validity of our results in Section 5. We conclude our work in Section 6.

2. RELATED WORK

This section presents the related work regarding the benefits of deploying design patterns and regarding change tactics.

2.1. The Benefits of Deploying Design Patterns

There are a number of empirical studies on the various issues related to the deployment of design patterns. These include whether the deployment of design patterns could reduce the occurrence of faults in a program, build a stable program in face of changes, and ease the understanding of software design. The issue related to the work experience of maintainers, however, has not been studied.

Vokáč [28] investigated, in the same C++ program, whether classes participating in design patterns are less faulty than those not participating in any design patterns. Vokáč studied the *Decorator*, *Factory Method*, *Observer*, *Singleton*, and *Template Method* patterns. The results showed that codes designed with the *Observer* and *Singleton* patterns were faultier, and the codes designed with the *Factory Method* pattern were less faulty. No conclusion could be drawn for the *Decorator* and *Template Method* patterns. Vokáč concluded that the deployment of design patterns does not guarantee fewer faults for classes participating in design patterns. However, with reference to the real-life situations where faults exist in typical industrial software products, he observed that codes designed with the *Observer* and *Singleton* patterns were areas with irreducible, significant complexity. Thus, these codes warrant special attention in design and implementation.

To evaluate whether programs having design patterns would be less subject to changes, and hence intuitively more stable, Bieman *et al.* [6][7] performed two studies to examine that, when design patterns were deployed, whether classes that participated in design patterns would have lower change frequency than the other classes. Five (C++ and Java) software systems, with one up to 37 versions, were examined. Twelve design patterns were examined in total. Among the five systems, four indicate negative outcomes and one presents positive outcomes. These results [6][7] confirmed that the deployment of design patterns cannot facilitate changes of all possible types [9]. Gamma *et al.* [12] also expressed that each design pattern may only be useful for specified, not arbitrary, maintenance problems.

Conventional wisdom suggests that the deployment of design patterns would result in additional flexibility [8]. Prechelt *et al.* [21] (in 2001) and Vokáč *et al.* [29] studied whether deploying a design pattern would still be useful when it is not directly related to an immediate maintenance problem. The deployment of the *Abstract Factory*, *Composite*, *Decorator*, *Observer* and *Visitor* patterns was investigated. In each study, both positive and negative scenarios were found. However, the reported results of two studies are conflicting. Prechelt *et al.* [21] discovered a negative scenario on deploying the *Observer* pattern and a positive scenario on deploying the *Decorator* pattern, but Vokáč *et al.* [29] reported a contrary result. Vokáč *et al.* [29] concluded that the question of whether the deployment of design patterns is beneficial cannot be generally answered, but depends on which problems and the way design patterns are deployed.

Our work is closely related to the above two studies [21][29]. Like theirs, our experiment assigned human subjects to perform change tasks on two functionally equivalent programs, with and without artificial instrumentation of design patterns. Unlike theirs, while the two studies focused on whether deploying a design pattern is beneficial (in terms of time spent and functional correctness) when it is not directly related to the target change tasks, our experiment focused on whether refactoring a program using additional design patterns to manage the anticipated changes related to the design patterns supports faster maintenance by less experienced maintainers. The two studies [21][29] also compared a program having no design patterns with a refactored program comprising at most two design patterns. This is a zero-to-many comparison. In our experiment, we compared a program comprising a dozen of design patterns with its refactored version instrumented with additional design patterns. Last, but not the least, the two studies ensured that subjects in different working groups have similar technical backgrounds. They examined the usefulness of using different design patterns in different change tasks. In contrast, our experiment assigned those subjects with and without work experience into distinct groups. We studied the dominance between work experience and refactorization using design patterns. Hence, our experiment is original and complementary to these two studies.

Prechelt *et al.* [20] (in 2002) evaluated whether explicit documentation of the deployment of design patterns would ease maintainers to perform change tasks in terms of time spent and functional correctness. The deployment of the *Composite*, *Observer*, *Template Method*, and *Visitor* patterns was investigated. Their results show that, with the support of the explicit documentation of deployed design patterns, maintenance is completed faster or with fewer faults. In our experiment, since the original deployed patterns in *JHotDraw* have been documented in the open source repository (Source Forge [25]), the documentations are explicitly passed to the subjects. However, to avoid the potential effect of pattern documentation on the instrumented design pattern in the refactored version, we did not explicitly inform the subjects about the deployment of the instrumented design patterns. As a result, the subjects needed to discover by themselves which deployed design patterns support their change tasks at hand. Further details of materials presented to the subjects will be described in Section 3.9.

2.2. Change Tactics

Let us overview the types of change tactics. According to Bass *et al.* [2], three types of change tactics may be used to facilitate software changes. They are localizing changes, preventing ripple effect and deferring the binding time.

Table 1. Quantitative comparison of our approach and related work

Study	Description metrics of the target program of maximum size	Type of target programs used	Number of subjects
Prechelt *et al.* [21] and Vokáč *et al.* [29]	683 LOC in 13 classes	Self-developed	≤44 human subjects
Prechelt *et al.* [20]	560 LOC in 11 classes	Self-developed	96 human subjects
Vokáč *et al.* [28]	505367 LOC in 2047 classes	Industrial	153 program revisions
Bieman *et al.* [6][7]	753000 LOC in 7573 classes	Industrial	39 program revisions
Our approach	14342 LOC in 207 classes	One open source and one refactored from the open source version	118 human subjects

In the object-oriented paradigm, a design pattern factorizes its solution of a recurring problem into separate classes, known as *participants* [12]. Each participant is a class whose details are hidden and encapsulated. Some design pattern supports the addition of functionalities as subclasses of these participants. In this way, a design pattern localizes changes directly, which is also one of the objectives of these design patterns [12]. The basic mechanisms of design patterns are the use of abstract interfaces to confine the ripple effects of change, and the use of dynamic binding and polymorphism to defer the binding decision between a caller object and an object being called.

In addition to these direct benefits of object-oriented techniques, design patterns are characterized by their provision of *hooks* [12]. Hooks in design patterns aim to collectively facilitate maintainers to easily add new classes that behave like target participants of the design patterns. Following up this concept of hook [12], we concretize a hook to be a (expected) method of a participant (i.e., a class), which implements specified sequences of method invocations as stated in the concerned design patterns. For example, the *Composite* pattern [12] defines a unified interface for both primitives and their containers, and let subclasses define their respective specific implementations. The *Composite* pattern gives subclasses hooks for providing new kinds of primitives and containers [12], because newly defined subclasses can integrate seamlessly with the existing participants of the design pattern. Clients do not need to be changed for the newly defined subclasses. The *Factory Method* pattern defines an interface for creating an object, but let subclasses decide which class to instantiate. Similar to the hooks provided by the *Composite* pattern, the *Factory Method* pattern gives subclasses hooks to provide new kinds of the objects to be created [12].

3. METHODOLOGY OF EXPERIMENT

This section presents the planning and operation of our experiment.

3.1. Hypothesis

If a deployed design pattern provides hooks to perform a change task, we say that the design pattern *supports* the task. In this experiment, we studied the following two approaches to perform a change task that is not supported by any design patterns in the original program:

- **Direct Approach**: When the change task is needed, maintainers perform the change task by directly revising the original program.
- **Refactoring Approach**: The original program has been refactored using additional design patterns to support the change task in advance. Maintainers perform the change task by revising the refactored version.

The hypothesis of this experiment is as follows:

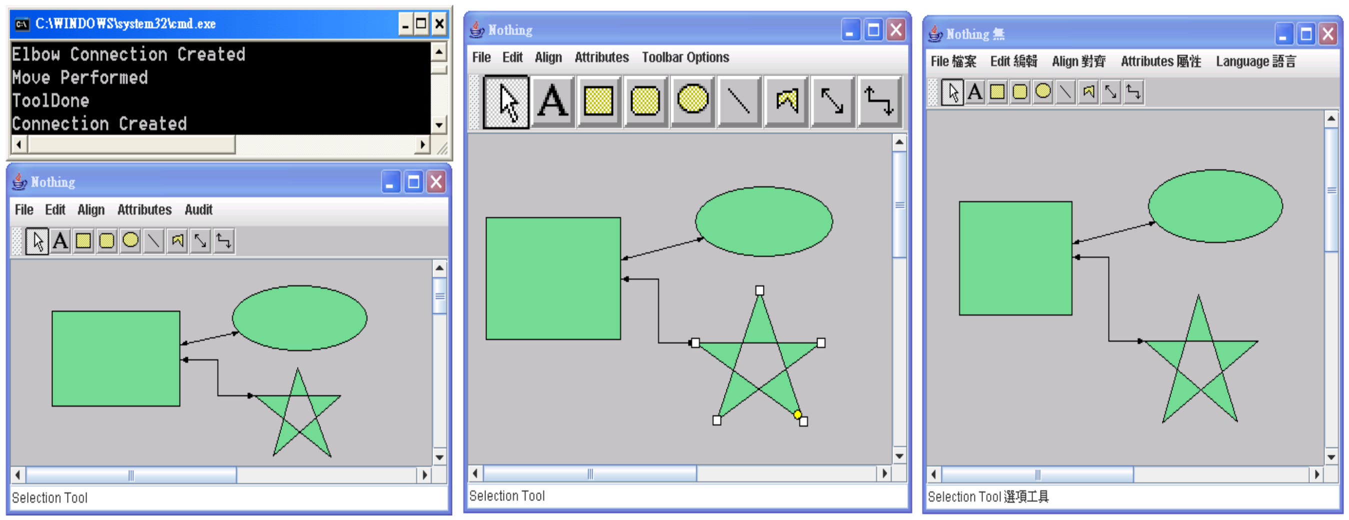
- **Null Hypothesis (H_0):** With regard to the same change task, there is no significant difference between using the direct approach and using the refactoring approach.
- **Alternative Hypothesis (H_1):** With regard to the same change task, there is a significant difference between using the direct approach and using the refactoring approach.

3.2. Requirements of Our Experiment

To ensure that the appropriate phenomena were studied, the methodology of our experiment needed to meet two specific requirements: realism and replication [24].

a) **Realism**. To test the hypothesis, we needed to study a situation, which is representative amongst realistic change tasks to be performed on software programs. In practice, we needed to find (i) a program large enough to be developed by a number of people over multiple evolution cycles, and (ii) some change tasks that are challenging enough to require maintainers to spend a significant amount of effort investigating the program.
b) **Replication**. The successful implementation of the selected change tasks could be heavily influenced by the nature of the task, the programming language used, and other factors independent of the maintainers, so we needed to control the change tasks and the tools in which the work is performed.

These requirements are however conflicting [24]. When we include more change tasks in the study, we have to control more factors that might influence the results. This makes it harder to collect sufficient amount of data for analysis that aims to explain the inherent complexity of the phenomenon to be observed [24]. In view of this, we settled on an experimental setting that could be favorably compared to those adopted by the related work. A comparison is given in Table 1. The comparison is based on the size of the programs studied, number of subjects, and design patterns involved. Since some studies investigated more than one target program, one comparison criterion is the size of the largest target program. Compared with the related work involving only human subjects, our experiment was conducted with more subjects and using a larger target program in size. In addition, amongst the listed work, our experiment is the only one using an open source system as a target program, whose results are amenable to be


C:\WINDOWS\system32\cmd.exe
Nothing
File Edit Align Attributes Audit
Selection Tool
Nothing
File Edit Align Attributes Toolbar Options
Selection Tool
Nothing 無
File 檔案 Edit 編輯 Align 對齊 Attributes 屬性 Language 語言
Selection Tool 選項工具

```
Elbow Connection Created
Move Performed
ToolDone
Connection Created
```

Figure 1. Screen shots of programs that fulfil the change tasks

replicated by others. Also, more design patterns are deployed in the target program than those reported in the related work. On the other hand, we had chosen three changes tasks to be performed on the target program and each of these change tasks requires us to spend at least three hours for completion. Based on this quantitative comparison, our experimental design contributes to an acceptable level of realism. More details about the target program and the change tasks are presented in the next two sections.

3.3. Target Program

In our experiment, we used the *JHotDraw* drawing editor [14] as a vehicle to test the above-mentioned hypotheses. *JHotDraw* is an open source system written in Java. Using the metrics plug-in of *Eclipse* for code analysis, the version we used (*JHotDraw* version 6.0-beta2) consists of 14342 noncomment, nonblank lines of source code, distributed over 207 classes in 10 packages. Up to this version, there have been 24 developers implementing *JHotDraw*. *JHotDraw* was registered at Source Forge [25], a popular repository of open source systems, as an open source system on 10th October, 2000. *JHotDraw* has been deployed with a dozen of design patterns described by Gamma *et al.* [12], namely *Adaptor*, *Command*, *Composite*, *Decorator*, *Factory Method*, *Mediator*, *Observer*, *Prototype*, *Singleton*, *State*, *Strategy*, and *Template Method*. The deployment of these design patterns is explicitly mentioned in *JHotDraw*'s Application Programming Interface (API) documentation. In addition, the preliminary version of *JHotDraw* was developed by the first author of [12]. As a result, *JHotDraw* is an excellent candidate to conduct design patterns research. In fact, *JHotDraw* has been used as an illustrative example or a case study in various existing work [1][23].

3.4. Change Tasks

The change tasks on *JHotDraw* employed in our experiment are described as follows:

a) **Audit**. A new menu "Audit" needs to be added. This menu would support a new option "Record Action History". After the option is enabled, log messages of the following actions would be displayed on the console (see Figure 1 (left)):
   - Creating a *figure* such as a text, rectangle, round rectangle, ellipse, line, polygon, and (elbow) connection.
   - Moving a created figure.
   - Performing a command such as cut, paste, duplicate, delete, group, ungroup, send to back, bring to front, undo, redo, align and change attribute.

b) **Image**. A new menu "ToolBar Option" needs to be added. This menu would support a choice of "Small" or "Large", determining the size of the icons displaying on the buttons of the tool bar (see Figure 1 (middle)).

c) **Language**. A new menu "Language" needs to be added. This menu would support a selection of "English", "Traditional Chinese", and "Simplified Chinese". When a language is selected, the text appearing in the following graphical component will be translated accordingly (see Figure 1 (right)):
   - The title of the software program
   - The menu bar
   - Tool tip text displayed in the tool bar
   - The status bar

All the three change tasks are perfective. In general, a change task can be adaptive, corrective, and perfective [18]. An adaptive task handles changes in the environment such as changes in the operating system and database management system, to name two. Unlike a corrective task which fixes bugs, a perfective task improves a system's emergent qualities such as performance, functionality, and maintainability. Our experiment focuses on perfective tasks because various studies [4][18] report that perfective tasks frequently occur in software lifecycles.

To test the hypotheses, we ensured that the three employed change tasks are not facilitated by the design patterns that have been deployed in the *JHotDraw* (version 6.0-beta2) program. To validate this, we identified in the program the following statements, which needed to be modified to carry out the change tasks, and checked if these statements play any roles in the documented design patterns of *JHotDraw*.

a) **Audit**. All statements that print a message to the console after the user performs an action that changes the drawing panel of *JHotDraw*.

b) **Image**. All statements that determine the physical location of an image file for an icon displayed on the toolbar.

c) **Language**. All statements that determine the rendering of texts for a graphical component.

3.5. Refactoring Process

The process of refactoring *JHotDraw* using design patterns covered three phases: selection of design patterns, restructuring of *JHotDraw*, and testing for functional equivalence [11]. To ease explanation, let us refer to the *JHotDraw* program before and after refactoring as the *original* and *refactored* version, respectively.

3.5.1. Selection of Design Patterns

The selection of design patterns was based on the types of design patterns and the occurrence of pattern names in academic literature. The first criterion ensures design patterns of different types of purpose are selected. The second criterion ensures that the chosen design patterns had some practical relevance.

The design patterns to which *JHotDraw* is refactored include the *Composite*, *Decorator*, *Factory Method*, and *Observer* patterns. *Composite* and *Decorator* are object structural patterns, *Factory Method* a class creational pattern, and *Observer* an object behavioral pattern. Vokáč [28] presented a ranking of design patterns according to the occurrence of pattern names. Our selected design patterns are the top four of the ranking.

3.5.2. Restructuring of *JHotDraw*

For each change task, we separately refactored the original version of *JHotDraw*. Table 2 summaries the refactoring procedures used to support each change task. The details of the selected procedures are presented by Kerievsky [16]. The goal of the refactoring is to support a change task so that the statements outlined in Section 3.4 for the change task are allocated in the concrete participants that represent hooks as described in the specification of the instrumented design patterns.

Table 2. Our refactoring procedures on *JHotDraw*

Change Task	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*

3.5.3. Testing for Functionally the Same

To validate whether the refactoring procedure preserves the functionality of the original programs, we have performed both category partition testing and code inspection. During category partition testing, we have ensured that the selected test cases cover the entire graphical user interface of *JHotDraw* for minimal functional operability. We used the original program of *JHotDraw* as the oracle. During inspection, we have ensured that the design patterns instrumented in *JHotDraw* conform to the structure and collaboration in their original specifications [12].

3.6. Subjects

There are two sets of subjects in our experiment. The first set consists of 55 subjects enrolled in an undergraduate-level Java programming course offered by the Hong Kong University of Science and Technology. These subjects were full-time undergraduates with no formal work experience in industry. We refer to this group of subjects as *inexperienced subjects.*

The second set comprises 63 subjects enrolled in a postgraduate-level software engineering course offered by the Hong Kong University of Science and Technology. These subjects have already obtained their first degrees in computer science or equivalent. They were full-time software developers with at least one year's work experience in the industry. We refer to this group of subjects as *experienced subjects.*

In addition to general work experience, one may compare subjects with and without knowledge about *JHotDraw*. For example, if a program is implemented in Java and designed using an object-oriented paradigm, the experience in Java and object-oriented concepts may affect the performance of subjects to maintain the program. To address this issue, we present a comparison of the programming background between inexperienced and experienced subjects in Table 3. The experienced subjects on average have much more work, Java programming and object-oriented design experience over the inexperienced ones. Further analysis could be done on whether the subjects are familiar with design patterns, the deployed design patterns or other factors. They are however not within the scope of this paper.

Table 3. Experience comparison between inexperienced and experienced subjects (in terms of year)

Experience	Inexperienced subjects	Experienced subjects
Working in the industry	Mean: 0	Mean: 5.66
	S.D.: 0	S.D.: 4.74
Java	Mean: 0.48	Mean: 3.25
	S.D.: 0.94	S.D.: 2.80
Object-oriented concepts	Mean: 2.13	Mean: 4.15
	S.D.: 1.01	S.D.: 2.92

3.7. Task Assignment

We divided subjects into three groups, and compared the experimental results at the group level (instead of the individual level). Groups 1, 2 and 3 consisted of 63 experienced subjects, 31 inexperienced subjects and 24 inexperienced subjects, respectively. Table 4 shows the allocation of change tasks on different program versions to subject groups. In the task allocation exercise, we ensured that groups with and without experience respectively performed one change task using the direct approach and another using the refactoring approach, so that the performance of different groups using different approaches can be compared. As shown in Table 4, there are two such comparisons.

Ideally, the experimental design should be a 2 × 2 full factorial design, that is, for each change task assigned to a subject group, the subject group is partitioned into two subgroups taking the direct approach and the refactoring approach, respectively. A limitation of our design as compared with this full factorial design is that fewer data points would be available for analysis. However, there was a pragmatic reason for our design. Our experiment was carried out as part of two software engineering classes, one with experienced subjects and another with inexperienced subjects. In each class, we abided by a course policy to conduct the same assessment of every subject's performance. One simple alternative is to assign experienced subjects to take the direct approach and inexperienced ones to take the refactoring approach for only one single change task. If the latter class of subjects performs better, one may conclude that program refactoring using additional design patterns is a more dominant factor than work experience. Nonetheless, this simple experimental design makes an assumption that experienced subjects must perform better than inexperienced ones; otherwise an interesting hypothesis would be difficult to formulate. To relax this assumption, we assigned experienced subjects to the refactoring approach and inexperienced ones to the direct approach for another change task. Furthermore, since we could split the inexperienced subjects into two groups, we repeated the above procedures using different change tasks for inexperienced subjects. This enabled us to set up an experiment to obtain more types of data points under our course policy.

Table 4. Task assignments to subject groups

Comparison	Task	Approach	Group
A	*Image*	Direct	1
		Refactoring	2
	Audit	Direct	2
		Refactoring	1
B	*Language*	Direct	1
		Refactoring	3
	Audit	Direct	3
		Refactoring	1

3.8. Variables

Three types of variables are defined for the experiment, *independent*, *controlled* and *dependent* variables.

a) **Controlled Variables**.
- *Eclipse* is used as the integrated development environment due to its popularity in industry to develop Java programs.
- *JHotDraw* is used as the program to be modified.
- Java is the programming language used, due to the selection of *JHotDraw* as the target program.
- Change tasks for *JHotDraw*: *Audit, Image* and *Language*. Although there are three change tasks, we compare the performance of different subject groups under different approaches with regard to the same task. As a result, change tasks are controlled variables.

b) **Independent Variables**.
- The direct and refactoring approaches.
- Subjects with and without work experience.

c) **Dependent Variables**.
- Time spent to complete a change task.
- Number of failed test cases.

3.9. Materials

For each task assignment, the materials presented to subjects to perform a change task on a program version were as follows:

a) **Requirements Specification**. Subjects were required to achieve functional correctness for the change task. In order to specify the functional requirements clearly, we used a demonstration program that implements the change task and deemed it as a functionally correct version. This program was presented to the subjects. To avoid plagiarism of the program, the program was 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 zero marks. With our analysis of the decompiled version, the effort needed to understand the decompiled version is more than that to modify the given version to complete a change task.

b) **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 *Eclipse*.

c) **Documentation**. The API documentation of the program to be modified was presented to the subjects. The UML models of each class hierarchy generated by reverse engineering tools were also presented to the subjects.

3.10. Experiment Procedures

To prepare the subjects for the task assignments, one week was devoted to education. This consists of two one-hour information sessions for each group separately. The first information session was a tutorial to use *Eclipse*, to ensure they could compile and run the program to be modified successfully themselves. The second information session explained the task assignments assigned to them.

During the information session, we executed the demonstration program to describe the functional requirements. We also showed the API documentation and class hierarchies of the program to be modified, where the API documentation mentions the design patterns deployed in the open source version of *JHotDraw*. As each group was to perform different tasks on different program versions, we emphasized that the programs to be modified for different tasks were different and reminded them to perform each task based on the given program version.

We did not explain the details of each design pattern to the subjects, but presented the general object-oriented concepts. We did not explicitly inform those subjects using the refactoring approach about any instrumented design patterns. One may argue that it is unfair not to present this information to the subjects, but there were two reasons for this. Firstly, as mentioned in Section 2, we wished to avoid any explicit pattern deployment affecting our results. Secondly, this matches the realistic situations where documentation often becomes non-synchronized after software changes.

Each task assignment was carried out individually by each subject. The subjects were given one month to complete the assignment. They were also requested to report the time spent on their assignment.

4. RESULTS

This section presents the data collected during the experiment and analyzes the performance of the subjects. The measurements are the time spent, the number of failed functional test cases, and the time spent regarding only those submitted programs with no failed functional test cases found. In particular, in the direct approach, the time spent is the time taken by the subjects to revise the original version of *JHotDraw*. In the refactoring approach, the time spent is the time taken by subjects to revise the refactored version of *JHotDraw*, plus the duration of the refactoring process for the corresponding change tasks (Table 5 reports our self-recorded time to refactor *JHotDraw* for each task). For each measurement, a descriptive analysis is firstly presented, followed by the associated statistical analyses.

Table 5. Time to refactor *JHotDraw* for each change task

Change Task	Time to Refactor
Audit	3 hours
Image	5.5 hours
Language	4.5 hours

Our collected data show that some subjects did not submit their work by the deadline or report the time spent. Table 6 shows the ratio of the number of submitted programs and that of those reports which include the time spent against the total number of subjects participating in each task assignment (c.f., Section 3.7). In the subsequent analyses, we excluded from the data associated with those subjects who reported only the time spent but did not submit any programs. Statistical analysis was conducted based on the 5 to 95 percentiles of the processed data. As such, the amount of data processed in the subsequent analysis will be approximately 90% of those reported in Table 6.

Table 6. Number of submitted work and reports on time spent to total number of subjects in each task assignment

Task-Approach-Group	Number of Submissions	Number of Time Reports
Image-Direct-1	59 out of 63	54 out of 63
Image-Refactoring-2	24 out of 31	24 out of 31
Audit-Direct-2	30 out of 31	28 out of 31
Audit-Direct-3	23 out of 24	23 out of 24
Audit-Refactoring-1	62 out of 63	62 out of 63
Language-Direct-1	62 out of 63	61 out of 63
Language-Refactoring-3	21 out of 24	21 out of 24

4.1. Time spent

Table 7 gives the number of collected programs, the means and standard deviations of the time spent. Figure 2 presents a box-and-whisker plot on the time spent on those task assignments applicable to *Comparison A*, which compares Group 1 and Group 2 taking the direct and refactoring approaches with regard to the same change task. For the *Image* task, when Group 1 took the direct approach, Group 2 took the refactoring approach. For the *Audit* task, the approaches taken by the two groups were switched. Figure 3 depicts the results for *Comparison B*, exhibiting similar results as in Figure 2.

The nonparametric nature of the data implies that the Mann-Whitney test is applicable for hypothesis testing. Table 8 gives the test results on the null hypothesis described in Section 3.1, which measures the time spent by various groups conducting the same task using different approaches. Each test produces a result at the five-percent statistically significance level. Together with the descriptive data in Table 7, we conclude that the refactoring of a program using design patterns is a dominant factor over work experience of maintainers in terms of the time spent to complete a change task.

4.2. Functional Failures

We performed category partitioning on the submitted programs, and then selected test inputs from each partition to conduct full combination testing. We aimed to validate functional correctness.

a) *Audit*. There are two equivalence classes for the *Audit* task. The first equivalence class is whether the menu item "Record Action History" is checked. Another equivalence class is the execution of different commands displayed in the menu bar or the tool bar, or the moving of drawn figures. For each partition, there are 21 different kinds of scenarios. We have developed one test case for each setting of the equivalence classes and executed the test cases in random order. Totally, we have $2 \times 21 = 42$ test cases.

Table 7. Descriptive data on time spent on each task assignment (in terms of hour)

Task-Approach-Group	n	Mean	S.D.
Image-Direct-1	48	11.36	6.53
Image-Refactoring-2	22	7.70	1.75
Audit-Direct-2	26	11.39	6.07
Audit-Direct-3	21	8.08	4.22
Audit-Refactoring-1	56	5.86	2.22
Language-Direct-1	55	18.31	11.92
Language-Refactoring-3	19	7.82	1.42

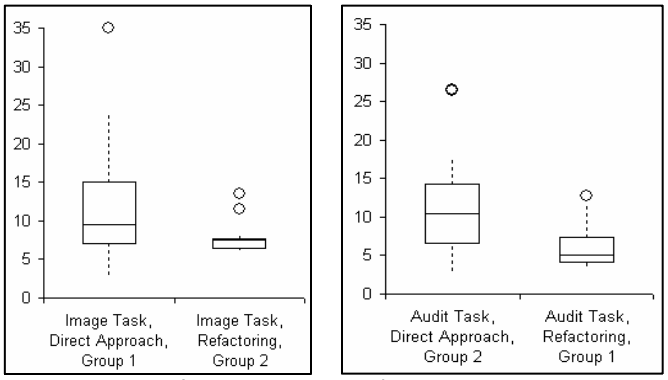


Figure 2. Graphical display of time spent regarding *Comparison A* (in terms of hour)

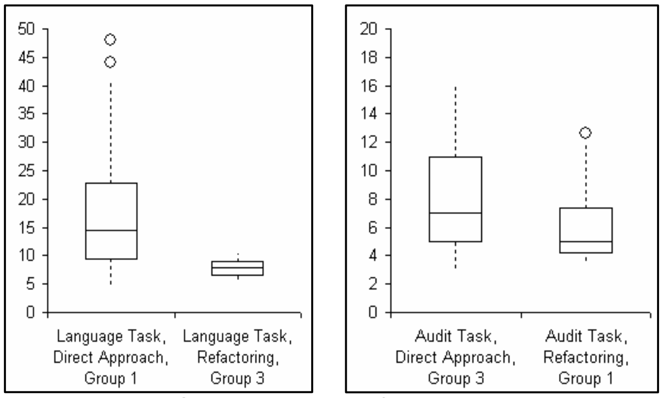


Figure 3. Graphical display of time spent regarding *Comparison B* (in terms of hour)

Table 8. Mann-Whitney test results on the null hypothesis in terms of time spent

Criterion	z	Conclusion
The null hypothesis in terms of time spent on the *Image* Task for Group 1 and Group 2	2.71	We **rejected** the null hypothesis in terms of time spent at 5% significance level ($z > 1.96$)
The null hypothesis in terms of time spent on the *Audit* Task for Group 1 and Group 2	4.83	
The null hypothesis in terms of time spent on the *Language* Task for Group 1 and Group 3	4.57	
The null hypothesis in terms of time spent on the *Audit* Task for Group 1 and Group 3	2.08	

b) *Image*. Three equivalence classes are constructed for the *Image* task. The first one is the number of times the toolbar option is changed. Three settings are defined for this equivalence class: 0, 1, or more. The second equivalence class is whether or not a button on the toolbar is pressed whenever the toolbar option is changed. The last equivalence class is whether a figure is created by pressing a button on the toolbar in the final selection of the toolbar option. In total, we have $3 \times 2 \times 2 = 12$ test cases.
c) *Language*. The *Language* task is associated with two equivalence classes. The first one is the set of all possible combinations of the language options selected. There are seven settings for this set. The second one is whether or not executing some commands displayed on the menu bar or tool bar or moving a drawn figure after the final selection of the language options. We have $7 \times 2 = 14$ test cases in total.

Table 9 shows the descriptive data on the number of functional failed test cases over a program (that is, functional failures) for each task assignment. For each of the *Image* and *Language* tasks, subjects without work experience maintaining a refactored version as the whole achieve a lower mean value of failures. For the *Audit* task, Group 2 outperformed Group 1, which in turn outperformed Group 3 in terms of the number of functional failures.

From the statistical test results presented in Table 10, we fail to reject the null hypothesis in terms of functional failures for three out of four tests. Together with the descriptive data as previously mentioned, we conclude that when using the refactoring rather than the direct approach, the number of functional failures after the completion of the task neither increased nor decreased.

No conclusion can be drawn on whether subjects with or without work experience delivered less error-prone programs, and whether using the direct approach or the refactoring approach to complete a change task is more preferable. Similar results were shown in Figure 4 and Figure 5.

4.3. Time spent regarding Functionally Correct Programs

In Section 4.1, we investigated the time spent on all submitted programs. This section analyzes the time spent regarding only those programs that passed all of our functional test cases. These programs are considered as functionally correct. While Section 4.1 provides an analysis of the programs submitted by the deadline that signals the end of a development phase, this section presents an analysis of the submitted programs with best functional quality.

Table 11 shows the descriptive data on the time taken to modify a given program correctly in each task assignment. Figure 6 and Figure 7 present the graphical plots of the time taken to correctly complete the task assignment for *Comparison A* and *Comparison B*, respectively. Table 12 shows the statistical test results for the null hypothesis in terms of the time spent regarding correct programs. From Table 12, we can conclude that the refactoring of a program using additional design patterns is a dominant factor over work experience of the maintainers in terms of the time spent to complete a change task without functional failures.

Table 9. Descriptive data on number of functional failures for each task assignment

Task-Approach-Group	n	Mean	S.D.
Image-Direct-1	53	2.49 out of 12	2.85
Image-Refactoring-2	22	0.55 out of 12	1.87
Audit-Direct-2	28	1.36 out of 42	4.89
Audit-Direct-3	21	0 out of 42	0
Audit-Refactoring-1	56	0.46 out of 42	2.43
Language-Direct-1	56	6.29 out of 14	5.49
Language-Refactoring-3	19	3.58 out of 14	5.48

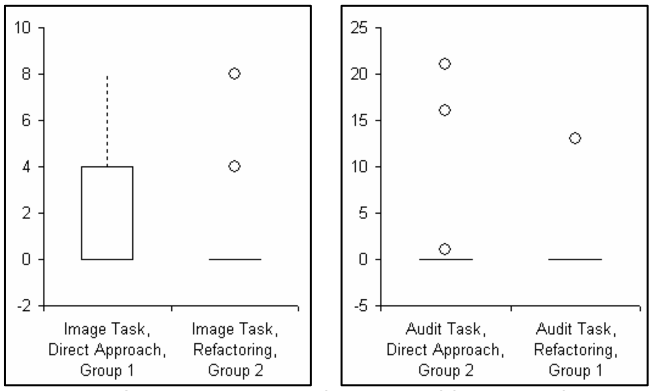


Figure 4. Graphical display of number of functional failures for *Comparison A*

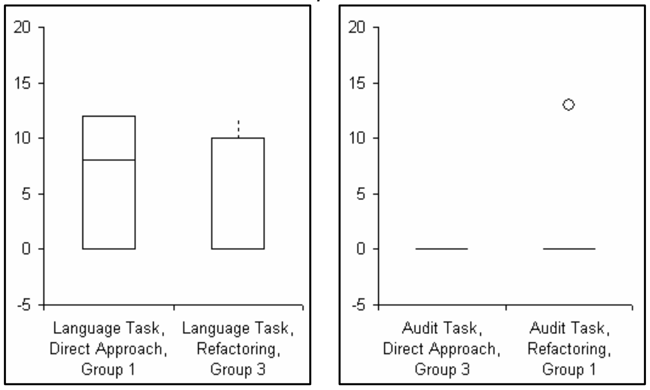


Figure 5. Graphical display of number of functional failures for *Comparison B*

Table 10. Mann-Whitney test results on the null hypothesis in terms of number of functional failures

Criterion	*z*	Conclusion
The null hypothesis in terms of number of functional failures on the *Image* Task for Group 1 and Group 2	2.64	We **rejected** the null hypothesis in terms of failures at 5% significance level ($z > 1.96$)
The null hypothesis in terms of number of functional failures on the *Audit* Task for Group 1 and Group 2	0.54	We **failed to reject** the null hypothesis in terms of functional failures at 5% significance level ($z < 1.96$)
The null hypothesis in terms of number of functional failures on the *Language* Task for Group 1 and Group 3	1.57	
The null hypothesis in terms of number of functional failures on the *Audit* Task for Group 1 and Group 3	0.23	

Table 11. Descriptive data on time spent regarding correct programs for each task assignment (in terms of hour)

Task-Approach-Group	n	Mean	S.D.
Image-Direct-1	28	11.65	7.24
Image-Refactoring-2	20	7.7	1.83
Audit-Direct-2	22	10.87	5.58
Audit-Direct-3	21	8.08	4.22
Audit-Refactoring-1	51	5.64	1.95
Language-Direct-1	24	14.75	9.64
Language-Refactoring-3	13	7.67	1.46

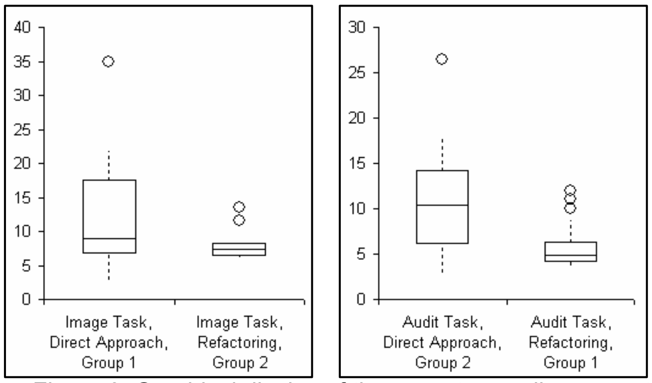


Figure 6. Graphical display of time spent regarding correct programs for *Comparison A* (in terms of hour)

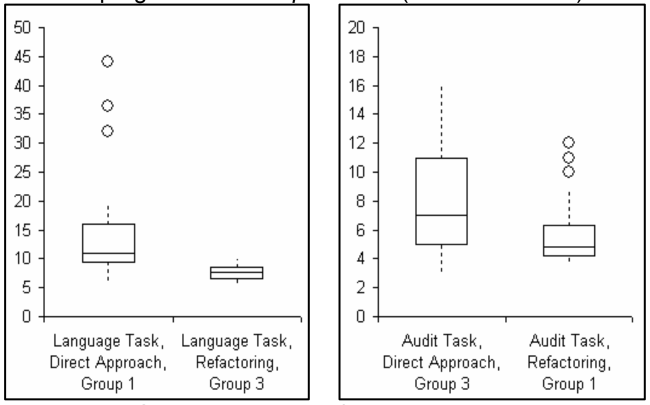


Figure 7. Graphical display of time spent regarding correct programs for *Comparison B* (in terms of hour)

Table 12. Mann-Whitney test results on the null hypothesis in terms of time spent regarding correct programs

Criterion	z	Conclusion
The null hypothesis in terms of time spent regarding correct programs on the *Image* Task for Group 1 and Group 2	2.3	We **rejected** the null hypothesis in terms of time spent regarding correct programs at 5% significance level ($z > 1.96$)
The null hypothesis in terms of time spent regarding correct programs on the *Audit* Task for Group 1 and Group 2	4.51	
The null hypothesis in terms of time spent regarding correct programs on the *Language* Task for Group 1 and Group 3	3.56	
The null hypothesis in terms of time spent regarding correct programs on the *Audit* Task for Group 1 and Group 3	2.29	

4.4. Further Analysis and Overall Results

Apart from the statistical analysis, we also performed a manual inspection on all submitted programs. From all submitted programs using the refactoring approach, we observed that the subjects made use of the hooks provided by instrumented design patterns to complete the change tasks at hand. In contrast, from the submitted programs using the direct approach, the subjects employed their own techniques to perform the change tasks. Since the instrumented design patterns have supported the required changes, the subjects using the refactoring approach saved more time when collecting the codes to be modified to perform the change tasks. Thus, the observation from our manual inspection is consistent with our experimental results regarding the shorter time spent on using the refactoring approach.

The interesting part of our results is that regardless of the work experience of the subjects, the time spent on using the refactoring approach is much shorter than that of using the direct approach even after including the duration of our refactoring process in the refactoring approach. Moreover, the correctness of their delivered programs does not significantly differ.

5. THREATS TO VALIDITY

Several factors can affect the validity of our experiment.

5.1. Construct Validity

Construct validity refers to the degree of which our experiment is measuring is what it is purported to measure.

One may query why the refactoring process is performed by us rather than the subjects using the refactoring approach, where the latter choice may also be a viable alternative. We have two reasons for this decision. Firstly, by doing so, we could control the number of independent variables to a manageable size. Secondly, compared with the subjects, we are more knowledgeable about *JHotDraw* because of our expertise in design patterns and our efforts in refactoring *JHotDraw*. We thus examine whether refactoring a program using additional design patterns to support a change task by experts of the program is beneficial. This kind of refactoring is particularly useful to companies with high turnover rates of software maintainers. In such a company, an expert of a program is less likely to be available when a change task anticipated before is really needed. Our results show that proactive refactoring using design patterns pays off in terms of the time it would take even a non-experienced maintainer to perform the change task in the future.

Another issue is the power of this experiment, which refers to the probability of rejecting a hypothesis when it is false. This statistical information indicates the likelihood of obtaining desirable results from this experiment. In this experiment, for each task assignment, we have one group of at least 24 subjects and another of 63 subjects. According to Cohen [10], a medium effect size of 0.5 was assumed. The significance level is taken at 5% [17]. The power for our experiment is 0.53. With a power level of 0.8 as recommended by the empirical software engineering community [17], the number of subjects required in each group would have to be 64. A potential issue with the experiment is lack of power, but we have ensured that the power of our experiment is favorable compared to the related work in [20][21][29].

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

A criticism on the internal validity is the plagiarism problem of the subjects. In the experiment, we found that the subjects generally asked questions actively during the information sessions. Before the assignment deadline, the subjects kept asking for clarification by emails. We believe that the subjects generally completed the tasks by themselves. Also, data analysis below the 95 percentile of number of functional failures revealed further eliminates those who copy their work from smarter subjects. Moreover, we did not find high degrees of code similarities across submitted programs with functional failures using plagiarism checking tools.

Learning effects of subjects is also a potential threat. Each subject group received three change tasks on different program versions of *JHotDraw*, but all the program versions actually have a reasonable amount of overlapping codes, especially those not participating in design patterns. This implies that after the subjects performed the first change task, the effort to understand the overlapping part of the codes for the remaining tasks was reduced. To address this, we have informed the subjects to explicitly obtain a general code understanding before performing any tasks, and then to count this effort towards the time spent on every change task.

Another concern is that the subjects may give up the change tasks when they find them too difficult or take too long to complete. For each group performing each task, at least 80% of the subjects submitted their assignments (see Table 6).

The quality of the refactoring process also counts. We have carefully selected the design patterns in our experiment, requiring the selected design patterns to cover different pattern types and be relevant practically. After pattern selection, the refactoring process is based on step-by-step instructions determined by Kerievsky [17], an expert in program refactoring using design patterns. We have made the best effort to test the functional equivalence between the original and refactored programs.

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* is used as the code base for change tasks. Different results might be obtained from using different programs with different requirements and natures. Also, in *JHotDraw*, the design patterns deployed are not exhaustive. Different design patterns have characteristics that lead to distinctive pros and cons. However, we have focused on the use of hooks of design patterns in our experiment. Future work will be in the generalization of using hooks of design patterns.

Another threat to the generality of our study is the use of only three perfective change tasks in our experiment. Although our study involved three nontrivial tasks requiring maintainers to investigate different system aspects (such as control-flow, state transitions, event handling), there exist many different software modification types. For example, a task can be adaptive, perfective or corrective. Clearly, each type has distinct characteristics. Still, our selected change tasks being so large that it cannot be completely understood in a short amount of time contributes to achieving an acceptable level of external validity.

Only *Eclipse* is used exclusively as the software development tool and the programming language used is fixed as Java. These additional factors limit the generality of the study to similar conditions Nevertheless, Java is a popular object-oriented programming language used in the market. *Eclipse* is a widely used tool in the industry to develop Java systems.

5.4. Reliability

An open-source code base was used as our testbed. The change tasks and our refactoring process were defined in detail. The complete experimental materials can be obtained from http://www.cs.ust.hk/~cssam/FSE/06.html, including the refactored versions of *JHotDraw* and the demonstration programs presented to the subjects. Thus, our reported study is replicable.

Another threat is the reliability of work experience and the time spent reported by subjects. We observed that the work experience reported by each subject largely matches the age of the subject minus the subject's bachelor-degree graduation year. On the other hand, we also recorded our own time spent to prepare for the demonstration binary codes presented to the subjects as the reference implementation for each task. Considering that we are experts in design patterns and knowledgeable maintainers of *JHotDraw*, we compare our work time with the subjects in Group 1, those with work experience. We found that our work time falls within the 25 to 75 percentile of the time reported by Group 1. From the collected data, the standard deviations of the time reported across different groups are compatible, so we reasonably believe that the time reported reflects the actual time they spent on the assignments. Finally, we conducted our descriptive and statistical analysis based on 5 to 95 percentile of time spent. This has further improved the reliability of the reported time spent on tasks.

6. CONCLUSIONS

Many design patterns are popular tactics to accommodate anticipated design changes. They provide hooks for maintainers to introduce new functionalities. It is thus attractive to build the software to have such or similar types of flexibility. However, while refactoring a program using more design patterns to support an anticipated change appears to be an obvious choice to support additional flexibility, the process of refactoring and revising the refactored programs may introduce faults into the program. To investigate this tradeoff, we have reported a controlled experiment in this paper. Our findings show that a perfective change task can be completed much faster using the refactoring approach. This result is robust regardless of the work experience of the subjects. We have also found that the number of functional failures is largely independent of whether or not the program has been refactored using additional design patterns to support the change task. It provides solid evidences to employ design patterns technology in software development.

In the future, we will further investigate whether developers with and without work experience may make the same types of mistakes or not. In particular, intuitively, novice programmers will produce more diverse types of faults than experienced programmers, because experienced programmers should have learnt from their experience to avoid some poor styles of programming and hence tend not to introduce certain type of faults.

Future work will also study the following research questions: Are there any special circumstances where careful program refactoring using additional design patterns still results in undesirable consequences in maintenance? Is program refactoring using additional design patterns beneficial to long-term maintenance that involves extensive revisions? What are the general criteria to judge whether a given refactoring task is cost-effective or optimal?

7. REFERENCES

[1] 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, Portland, Oregon, USA, Mar. 2003, pp. 352–362.

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

[3] 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.

[4] S. Belmonte, J.G. Consuegra, J.L. Gavilan, and F.J. Honrubia, "Development and Maintenance of a GIS Family Products: A Case Study", *Next Generation Geospatial Information - 2003* (NG2I 2003), Cambridge, Massachusetts, USA, Oct. 2003.

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

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

[7] 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, Sydney, Australia, Sep. 2003, pp. 40–49.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[9] M.P. Cline, "The Pros and Cons of Adopting and Applying Design Patterns in the Real World", *Communications of the ACM*, 39(10):47–49, 1996.

[10] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 1988.

[11] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

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

[13] J2EE EJB, http://java.sun.com/products/ejb/. (Last accessed: 3 Apr 2006)

[14] JHotDraw, http://sourceforge.net/projects/jhotdraw/. (Last accessed: 3 Apr 2006)

[15] JUnit, http://junit.sourceforge.net/. (Last accessed: 3 Apr 2006)

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

[17] B.A. Kitchenham, S.L. Pfleeger, L.M. Pickard, P.W. Jones, D.C. Hoaglin, K.E. Emam, and J. Rosenberg, "Preliminary Guidelines for Empirical Research in Software Engineering", *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002.

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

[19] Mircosoft COM, http://www.microsoft.com/com/. (Last accessed: 3 Apr 2006)

[20] 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.

[21] 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.

[22] J. Rajesh, and D. Janakiram, "JIAD: A Tool to Inter Design Patterns in Refactoring", in *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (PPDP 2004), ACM Press, Verona, Italy, 2004, pp. 227–237.

[23] M.P. Robillard, "Automatic Generation of Suggestion for Program Investigation", in *Proceedings of the 10th European Software Engineering Conference jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (ESEC/FSE-13), ACM Press, Lisbon, Portugal, Sep. 2005, pp. 11–20.

[24] 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.

[25] Source Forge, http://sourceforge.net/. (Last accessed: 3 Apr 2006)

[26] 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.

[27] T.H. Ng, and S.C. Cheung, "Proactive Views on Concrete Aspects: A Pattern Documentation Approach for Software Evolution", in *Proceedings of the 27th International Conference on Computer Software and Applications* (COMPSAC 2003), IEEE Computer Society Press, Dallas, Texas, USA, Nov. 2003, pp. 242-247.

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

[29] 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.