Toward Effective Deployment of Design Patterns for Software Extension: A Case Study *

T.H. Ng §
Dept. of Computer Science
City University of Hong Kong
cssam@cs.cityu.edu.hk

S.C. Cheung
Dept. of Computer Science
Hong Kong University of Science and Technology
scc@cs.ust.hk

W.K. Chan
Dept. of Computer Science
Hong Kong University of Science and Technology
wkchan@cs.ust.hk

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

ABSTRACT
A design pattern documents a reference design for the solution to a recurring problem encountered in object-oriented software development. The fundamental theme of design patterns is to encapsulate the concepts that vary. Software practitioners generally take it for granted that achieving the pattern theme would lead to extensible software. Is such a conjecture legitimate? A direct validation of the conjecture is difficult because it requires an objective measurement of extensibility, which is still an open controversial concept. In this paper, we examine the conjecture indirectly by exploring if there is any relation between the pattern theme and the Open-Closed Principle, which has been advocated by many to achieve extensible object-oriented software. We conducted an experiment based on 98 postgraduate students in a software engineering course. The experiment suggested that the satisfaction of the pattern theme generally lead to the conformance to the Open-Closed principle. However, three exception cases were found. We look into these few exceptions and provide some analyses.

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, Languages, Experimentation

Keywords
Software extension, design patterns, pattern deployments, Open-Closed principle

* 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.
§ Ng is a part-time PhD student in the Department of Computer Science, Hong Kong University of Science and Technology.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WoSQ'06, May 21, 2006, Shanghai, China
Copyright 2006 ACM 1-59593-085-X/06/0005…$5.00.

1. INTRODUCTION
Typical software applications have a number of releases ahead of their retirement. They need to maintain properly so that a subsequent evolution can easily modify a baseline version to fulfill the new or modified requirements [1]. Many software engineers bear in mind this type of maintenance consideration and design their software applications to allow further modification or extension. They usually (would like to) structure a program module so that the behavior of the module can be changed easily to accommodate future requirements. At the same time, they want the code base of the module protected from these future changes; otherwise, the program design will be easily ruined.

One of the most commonly adopted tactics is to apply the general design notion of *Open-Closed principle*, which means "software entities should be open for extension, but closed for modification" [13], in their designs. For software applications implemented in object-oriented languages such as Java, software designers frequently supplement their designs with design patterns [8]. Design patterns organize the design concepts against various fragments of the application code base systematically. They (e.g. [8]) have been regarded as successful artifacts for object-oriented software development [5][15][16]. Moreover, software designers can access many publicly available programs for reference. Despite many critiques of design patterns, all the above factors, amongst others, foster the adaptation of design patterns in typical software application development. In other words, it appears that software practitioners generally take it for granted that achieving the pattern theme would lead to extensible software.

We amongst others observe that it is non-trivial. First, although design patterns follow the idea of the Open-Closed principle to structure the *reference* solution of certain recurring problems [8], a software engineer (e.g. a programmer) may not constantly implement the desirable organizations of code base as if the software designers themselves implemented them. Moreover, the software engineer may not know clearly the intention of the software designers, despite the assistance of modern software design methodology and techniques such as the unified process and the UML [21]. They may split a decision into multiple design patterns. Similarly, they may confuse decisions and merge them into the same design pattern.

However, a direct validation of the above conjecture is difficult because extensibility is still under a hot debate. We resolve to examine the conjecture indirectly by exploring if there is any relation between the pattern theme and the Open-Closed Principle, which has been advocated by many to achieve extensible object-oriented software.

We propose a two-phase *scheme* to study the relationship between the pattern theme and the Open-Closed principle. First, a software designer mark fragments of a code base of an object oriented software application according to their design goals and classifies them as reusable or not. Next, a mechanical step will check whether the deployment of design patterns in the code base supports the Open-Closed principle.

To begin with, we first clarify what we mean by supporting the Open-Closed principle through design patterns. According to the basic idea of design patterns, code fragments could be classified as reusable or changeable. The reusable portions refer to those parts that execute similar scenarios generally without modifications. As an analogue, a client object may invoke objects from the same class hierarchy in the same way, even though the details of the objects from different classes of the class hierarchy actually differ. The changeable portions refer to the special portions that deal with certain specific problems. Using the above class hierarchy analogy, each subclass may override certain methods, or add new methods (or attributes) with respect to its ancestor class. These new (or override) methods (or attributes) are changeable portion. At their first sight, the distinction is obvious. It is however no longer apparent after a few maintenance cycles, in particular if these cycles involve design re-factoring.

We call the above classification *the fundamental theme* of a design pattern. We say that the deployment of design patterns in an object-oriented software application is said to *support* the Open-Closed principle if every reusable or changeable portion across and within the deployment are clearly separated from the non-reusable or invariant portions. We note that design patterns only document the reference solutions, which is abstract from the implementation sense, leaving software engineers pretty rooms to code their implementations. Therefore, although the skeleton appears to follow themes of the pattern, their body may violate the theme. An illustration example will be given in Section 2. In other words, our criterion is to check whether the code that ought to be within a particular design pattern is indeed encapsulated by that design pattern.

The main contribution of this paper is: It presents an empirical experimentation to evaluate whether the theme of patterns will result in a software application respecting the Open-Closed principle. Our experimentation shows a promising result.

The outline of this paper is as follows: Section 2 reviews design patterns and presents a motivation example. Section 3 describes the related work. Section 4 presents the design and results of our empirical study. Finally, there is a discussion in Section 5, followed by the conclusion in Section 6.

2. Preliminaries

This section presents background knowledge of design patterns together with a motivation example.

2.1. Design Patterns

A design pattern [8] is normally described by its name, intent, motivating examples of an object-oriented design problem, the situations under which the problem occurs, a proven solution to the problem, the pros and cons of the solution, examples of deploying the pattern and a set of related patterns. There are many variants, and they give different levels of detail.

The solution of a design pattern is typically expressed as structured prose and sketches such as class and sequence diagrams in object-oriented notation [8][19]. Object-oriented techniques [8] such as delegation and dynamic binding are utilized. The pattern prose describes what to be non-reusable and what to be reusable. It also describes how non-reusable portions are encapsulated to facilitate changes. This type of encapsulation follows the fundamental theme of design patterns, "encapsulating the concept that varies" [8]. This encapsulation promotes design reuse across the releases of the software application. It is commonly envisaged that by such organization of code base and design, software reuse could lead to substantial gains in productivity [10][14].

2.2. A Motivation Example

Example 1. Consider a multiple-user calendar manager (*MCM*) for appointment scheduling. During the scheduling of appointments, the manager displays additional verbose information to the program console for assisting error reporting and program debugging. Multiple verbose modes would like to be supported:

- *Null* – no irrelevant information is displayed,
- *Exception* – only error messages are displayed, and
- *Debug* – error messages and information related to database updates are displayed.

Software designers may want their design to be flexible enough to accommodate other verbose modes in addition to the three existing ones. However, these verbose modes may not be known until some later releases.

Suppose that the *State* design pattern is deployed by the software designer to prepare for the above kind of software extension. The operations participating in this deployment are shown in Figure 1. In this paper, we use UML class diagrams [21] to describe pattern deployments.

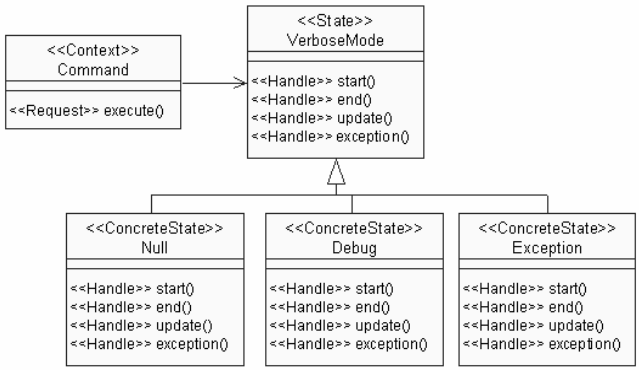

Figure 1: *State* pattern deployment in MCM (in UML notation)

2.3. An Undesirable Scenario

Figure 2 depicts a fragment of a Java [9] implementation of an operation responsible for adding new appointments into the calendar manager. It firstly starts (invoking start() of the State pattern) a verbose mode, and then poses any update (invoking update(appointment)) and finally ends (invoking end()) the verbose mode.

In this implementation, *S1*, *S2* and *S3* are fragments of the code base to support the debugging purpose. To ease our discussion in the rest of the paper, we call such a fragment of the code base a fragment "*specific to*" the *Debug* verbose mode. Similarly, *S4* is specific to the *Exception* verbose mode. During the program execution, the execution of these code fragments depends on the

```
public class InsertAppointment extends Command {
    ... // Other Operations

    public void execute(String[] args) {
            VerboseMode verbose = VerboseMode.getInstance();
            verbose.start(this);
            ... // Construct a new appointment object "appointment".

            String mode = MCM.getCurrentVerboseMode();
            if (Appointment.add(appointment)) { // if the addition is successful.
                verbose.update(this, appointment);
                if (verbose.getMode().equals("debug"))
        S1:         System.out.println("A new appointment is added: " + appointment);
                return appointment;
            } else { // if the addition is not successful.
                verbose.exception(this, appointment);
                if (verbose.getMode().equals("debug")) {
        S2:         System.out.println("The appointment already exists: " + appointment);
        S3:         Appointment.listAll(); // Print all existing appointments for debugging.
                } else if (mode.equals("exception"))
        S4:         System.out.println("The appointment already exists");
            }
            verbose.end(this);
    }
}
```

Figure 2: A client procedure that uses both the State Pattern and some "unclean" statements.

particular verbose mode in use. If the calendar manager supports a new verbose mode, the execution of these code fragments may need to be replaced by the code fragment specific to the new verbose mode.

In this regard, each code fragment specific to an existing verbose mode is not reusable to support the new verbose modes. On the other hand, all code fragments (except *S1*, *S2*, *S3* and *S4*) are reusable, because they are generally applicable to all existing (and new) verbose modes.
It is not hard to observe that the implementation of the operation outlined in comprises both reusable and changeable portions code fragments in the same procedure. As discussed above, this hampers the future support of potential verbose modes. Although a *State* pattern is deployed, the organization of code fragments does not support the Open-Closed principle in essence, defying the original purpose of applying the Open-Closed principle to construct extensible software.

3. RELATED WORK

In this section, we revisit the existing research work in expressing design patterns in certain design models of software applications, since a proper description of the pattern deployment may help alleviate the problem illustrated in Section 2. Existing works could be divided into informal and formal approaches. The most common category is to document the mapping between the object classes and the pattern solution of certain design patterns [4][7][17]. We discuss some representative ones. Jacobson *et al.* [4] document the object classes participating in a design pattern by means of class diagram and collaboration diagram in the UML notation. They do not describe how extensible software to be reasoned based on their approach. France *et al.* [7] document a similar mapping by stereotyping object classes as the roles of corresponding pattern participant. As a straightforward approach, their mapping is documented based on the original solution structure of the pattern. Hence, their results will change depending on the quality of the reference description of the design pattern. To resolve this shortcoming, they also include a meta-model, which expresses constraints in the deployed design patterns formally. Noda *et al.* [17] document such a mapping as a trace model between UML class diagrams.

Another category is to describe the deployment of patterns using formal specification techniques [7][20]. As discussed above, France et al. [7] specify pattern solutions as formal meta-models in UML [21]. When deploying a pattern, the class relationships and object collaborations that participate in a design pattern are expressed as UML class diagrams and sequence diagrams respectively. Structural and behavioural conformances are checked through the validation of UML models against UML meta-models.

In the motivation example described in Section 2, we have in fact ensured that structural and behavioral conformance is satisfied. However, the deployment of patterns in this example still does not support the Open-Closed principle. This shows that additional effort in addition to structural and behavioural conformance is needed to construct extensible software. In the next section, we show that such an additional effort is to satisfy the original theme of design patterns.

4. EMPIRICAL EXPERIMENTATION

This section presents an empirical experimentation to study the importance of satisfying the fundamental theme of design patterns which generally means "encapsulating the concept that varies" [8], in the deployment of patterns with regard to the Open-Closed principle.

Hypotheses: The goal of this experiment is to reject the following two hypotheses:

a) *When the theme of patterns is not satisfied, there is at least 20% of chance*[1] *where the Open-Closed principle is followed.* This hypothesis evaluates the necessity of satisfying the theme for conformance to the Open-Closed principle.
b) *When the theme of patterns is satisfied, there is at least 20% of chance where the Open-Closed principle is violated.* This hypothesis evaluates the sufficiency of satisfying the theme for conformance to the Open-Closed principle

1 The setting of the chance as 20% indicates that the theme of patterns is 80% close to necessary to establish conformance to the Open-Closed principle in a pattern-based program. This setting is a rough estimate due to the lack of similar study in the literature.

Participants. Ninety-eight participants were enrolled in a software engineering course of the Hong Kong University of Science and Technology. The participants were part-time postgraduate students with an average of 5-years work experience in the computer science industry. Participants had a diverse background of computer science knowledge. We believe that the participant pool is representative to reflect the experience of junior software practitioners in Hong Kong. They had a reasonable amount of experience of programming, but some of them had limited experience of UML, design patterns, and Java [9]. The experiment was being carried out as a part of a software engineering class, which counted as a part of their marks toward their degrees. Since they are mature and well-educated, we assumed that they conducted the experiment seriously.

Materials. The program under study in the exercise was MCM, the case study in this paper. The MCM program was implemented in Java [9]. The program implementation can be downloaded from [12]. Figure 3 presents a screen shot of the MCM program with an extension of a GUI. The materials presented to the participants for performing the exercise were as follows:

a) **Requirements Specification.** The requirements of the exercise are three-fold: (i) functional correctness, (ii) the deployment of State pattern, and (iii) the satisfaction of the fundamental theme of design patterns. Although our goal of this experiment was to evaluate the importance of the theme, functional correctness is essential for typical software development. As a result, functional correctness was included as a requirement of the exercise. Functionally, a MCM program as described in Example 1 had to be implemented.

   The second requirement required that the *State* pattern had to be deployed according to the design in Figure 1.

   As discussed in previous sections, even when the second requirement was fulfilled, the Open-Closed principle could still be violated. The third requirement required the satisfaction of the theme of patterns for supporting new verbose modes.

   To specify clearly the functional requirements, a demonstration of a model solution of the target program that was functionally correct was presented to the participants. To avoid plagiarism of the demonstration, the demonstration was in the form of obfuscated Java code. Since we controlled the time for the participants to walk through the code to iron out the genius part, we believed that the participants were able to understand the functional requirements but the satisfaction of the fundamental theme of patterns from the obfuscated code base. We also restricted the access of the code base so that no participant could make a copy and re-engineer a nice one from the obfuscated one.

b) **Source Code.** The source code presented to participants for revision was approximately 1500 lines of code (LOCs), while that of the final model solution contained about 2500 LOCs. Java was chosen for the source code documents to ease participants to focus on the requirements, rather than other issues that are not the common baselines amongst participants. This reduced the impact of learning effect and the differences in performance due to these anticipated factors.

c) **Documentation.** Participants were informed about the design issues related to the MCM program, including the system design model and the structure and rationale of pattern deployments. In addition, we give information to participants to simulate the fact that software designers would communicate with the programmers to pass the idea of the software designs of software applications.

Power. The power of an experiment is a standard statistical terminology, which denotes the probability of rejecting the hypothesis when it is false. This statistical information indicates the likelihood of obtaining desirable results from this experiment.

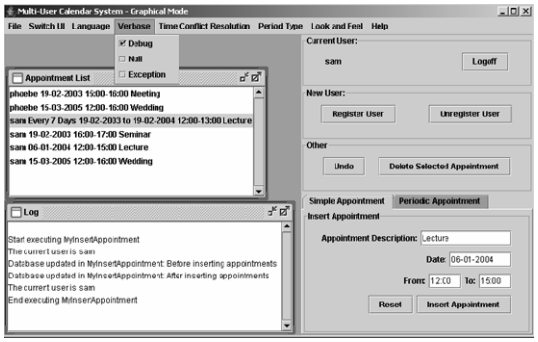

Figure 3: Screen shot of the MCM program with a GUI

In this experiment, since we made no assumption about whether or not the fundamental theme of design patterns would be more likely to result in a software respecting the Open-Closed principle, we therefore used a two-tailed test. We choose chi-square test (that is, χ^2 test) to analyze the data for the hypotheses. In line with Cohen [6], a medium effect size of 0.3 was used. The significance level was taken at 5% [11].

Since the sample size for the experiment was 98, for one degree of freedom, the power of the experiment was found to be 0.84. It satisfies the minimum power (at least 0.8) required by the empirical software engineering community [11].

Data Collection. Participants were asked to submit the source codes of their revised programs. They were also requested to submit a written report answering the following three questions regarding the experimentation. They were encouraged to make their own observations and to draw their own conclusions in the reports. The goal of these questions was to prepare for assisting the analysis of data after the program submission. It also helped check if there would be any abnormality against our hypotheses.

a) What would be the difficulty in achieving functional correctness?
b) What would be the difficulty in satisfying the theme of patterns?
c) How do you utilize the documentation in this exercise?

Experiment Procedures. We lectured every participant about the concept of UML and design patterns. In the information session, the participants read the requirements specification. The instructor also explained the experimentation to the participants. In particular, the instructor explicitly reminded them that when deploying patterns to construct extensible software, one's objectives should not only ensure functional correctness and the use of design patterns in their programs, but also satisfy the theme of patterns.

In case of uncertainty, participants asked questions about the experimentation and the instructor gave a general guideline without hinting how a particular issue could or could not lead to the construction of extensible software, in the sense of the fundamental theme of design patterns. As discussed above, the model solution needed to be extended with around 1 000 LOCs,

the participants were given five weeks to construct a target program based on the aforementioned baseline version. The grading criteria were set to 50% for functional correctness and 50% for satisfying the theme of patterns and the deployment of design patterns.

Threats to Validity. As in any empirical study, this experimentation exhibited a number of threats to internal and external validity. Internal ones included:

a) **Limited Java Knowledge.** The results of this experimentation might not be able to repeat if more experienced software engineers construct similar experiments.
b) **Plagiarism.** Participants might communicate with one another. This results in some hidden co-relations amongst some of the constructed programs. We had exercised measures to work against potential plagiarism in the experimentation.
c) **Distraction of Participants.** Participants were part-time students and most of them had full-time jobs. They might be occupied with the project deadlines of their full-time work. They might be able to construct quality solutions if time was allowed. Hence, the results of the experimentation should be interpreted conservatively.

As for the threats to external validity, participating entities in this exercise were not representatives of their respective general classes. For example, participants were not representative of the worldwide software engineering population. Java is a general purpose language, but cannot represent other programming languages. MCM is not representative of all software programs.

Results and Analyses. Figure 4 presents the overall performance of participants in the collection of the programs constructed by the 98 participants. All participants successfully deployed the *State* pattern according to Figure 1. 43 participants successfully delivered software respecting the Open-Closed principle. This translates into 43% of the total population. . It shows that deploying design patterns would not lead to the conformance of the Open-Closed principle. Amongst these 43 participants, 40 of them successfully constructed programs that exhibit the fundamental theme of design patterns according to our criterion. This accounts for 41% of the total population, or 93% of the population that respect the Open-Closed principle. Table 1 presents the statistical information for the χ^2 test on the hypothesis (a) with a 20% chance of achieving conformance to the Open-Closed principle when the theme of patterns is not satisfied.

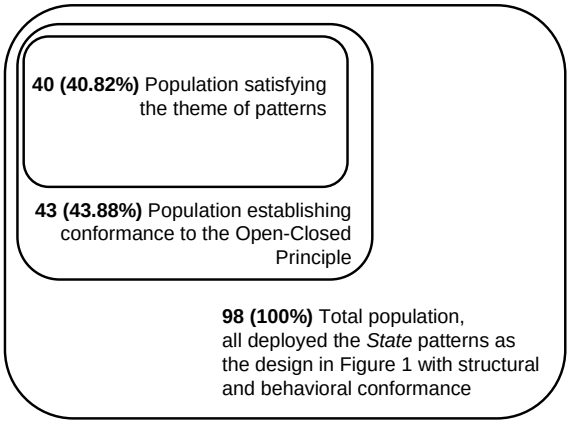

Figure 4: Performance of participants in Venn diagram

For one degree of freedom[2], a chi-square result of 7.07 was generated. Based on Table 1, if the chance of achieving conformance to the Open-Closed principle is set to more than 20%, an even larger chi-square result will be generated. This produced in a significant result at the one percent level (chi-square result > 6.64). Thus, we reject the hypothesis (a) at the 1% significance level. In other words, we conclude that when the fundamental theme of design patterns is not satisfied according to our criterion, there is less than a 20% chance of achieving conformance to the Open-Closed principle.

Table 1: Statistics for the χ^2 test on the hypothesis (a)

i	O_i	E_i	$\lvert O_i - E_i \rvert$	$(\lvert O_i - E_i \rvert - 0.5)^2$	$(\lvert O_i - E_i \rvert - 0.5)^2 / E_i$
a	3	11.6	8.6	65.61	5.66
$\neg a$	55	46.4	8.6	65.61	1.41
	58	58			$\chi^2 = 7.07$

Note: a denotes the condition of achieving conformance to the Open-Closed principle; i denotes the random variable "the condition of a"; O_i denotes the observed frequency; E_i denotes the expected frequency, as implied by the hypothesis (a) with a 20% chance of achieving a when the theme of patterns is not satisfied. Please be informed that there are 40 participants satisfying the theme, so the total observed (expected) frequencies are therefore 58 (= 98-40).

Similarly, we can reject the hypothesis (b). In fact, from Figure 4, no participant satisfying the theme of patterns while violating the Open-Closed principle. In other words, satisfying the fundamental theme of patterns would lead to a software application respecting the Open-Closed principle in a pattern-based program.

In Figure 4, there are three submitted programs that do not satisfy the fundamental theme of design patterns, yet respecting the Open-Closed principle according to our criterion. We examined their code bases. In each of these programs, there existed some operation specific to some verbose modes without participating in the *State* pattern deployment. All such operations were delegated from the operations that were specific to some verbose modes and participate in the pattern deployment. Nevertheless, we found that these delegations were actually irrelevant to the concept that should be controlled by the notion of the Open-Closed principle of the deployed patterns.

An interesting finding is that the participants reported that functional requirements and the deployment of the State pattern are relatively easy to meet as compared to the fulfilment of pattern theme. They seldom consider the effectiveness issue of pattern deployment in their daily work.

5. DISCUSSION

In this section, we discuss the issue related to the design patterns and Open-Closed principle. We have exploited the relation between design patterns and the Open-Closed principle as a mean to construct extensible software through an empirical experimentation in this paper.

[2] The degree of freedom equals the number of classes minus the number of restrictions in the χ^2 test. From Table 1, there are two possible values of i, implying that the number of classes is 2. There is one restriction, $\Sigma O_i = \Sigma E_i$. As a result, the degree of freedom is 1.

Ideal Open-Closed principle does not exist in real software applications. We do not assume such perfect scenarios in our experimentation. We only assume that the part being extended by the participants could be delivered through the application of the Open-Closed principle. In fact, we have implemented a model solution ahead of the participants doing the experimentation.

Apart from the Open-Closed principle, software designers may consider many other forces (e.g. the Liskov's Substitution Principle). The trade-offs amongst these principles are non-trivial. Certainly, these forces affect the structure of the programs. We plan to study this aspect in the future.

Furthermore, the deployment of patterns is known to not always yield good outcomes in terms of maintenance. In an empirical study, Prechelt *et al.* [18] conducted a controlled experiment to investigate if pattern deployments could facilitate program maintenance. In most cases, they conclude that design patterns improve the maintenance productivity. However, there are cases, which are not isolated ones, yielding adverse effects. In other words, design patterns are obstacles in maintenance in these non-isolated cases. In these few cases, the design solutions without deploying patterns were less error-prone or subject to shorter maintenance time.

Bieman *et al.* [3] conducted an industrial case study to explore the relationships between design patterns, design structures and program changes. Later, Bieman *et al.* [2] investigated five evolving programs to identify the benefits of deploying patterns for software design and maintenance. They [2] concluded that there is no solid evidence suggesting that the deployment of patterns practically facilitates software extension. These observations motivate the need to evaluate if design patterns are effectively deployed to facilitate software maintenance. This paper has reported an initial study towards answering this important question. It demonstrates the importance of following the theme of design patterns in the deployment of patterns. Further study is required to better understand the underlying issues of effective pattern deployment. In particular, we will observe the relation between the deployment of design patterns and functional errors.

6. CONCLUSION

Achieving flexible program structure to accommodate changes is attractive. Many software engineers consider that the general concept of Open-Closed principle may help them. They take design patterns for granted to assume that deploying design patterns will naturally lead to software, which is extensible. Our example shows a counter-example.

The main results show that the theme would lead to a software application respecting the Open-Closed principle in a pattern-based program. In addition, in the experimentation, violating the theme would likely lead to the software application disrespecting the Open-Closed principle. They are based on our criterion that checks whether the code that ought to be within a particular design pattern is indeed encapsulated by that design pattern.

Future research will proceed to develop tools to automate the checking of the fundamental theme of design patterns. More criteria of deploying patterns will be studied. We will study other forces and tradeoffs influencing the software design of extensible software.

7. REFERENCES

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

[2] 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", *Proc. 9th Int'l Software Metrics Symp.* (METRICS 2003), IEEE Computer Society Press, Sydney, Australia, Sep. 2003, 40–49.

[3] J.M. Bieman, D. Jain, and H.J. Yang, "OO Design Patterns, Design Structure, and Program Changes: An Industrial Case Study", *Proc. 17th Int'l Conf. Software Maintenance* (ICSM 2001), IEEE Computer Society Press, Florence, Italy, Nov. 2001, 580–589.

[4] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Software Development Process.* Addison Wesley, 1999.

[5] 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.

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

[7] R.B. France, D.K. Kim, S. Ghosh, and E. Song, "A UML-Based Pattern Specification Technique", *IEEE Trans. Software Eng.*, 30(3):193–206, 2004.

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

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*. 2nd Edition, Sun Microsystems, 2002.

[10] W.S. Humphrey, *A Discipline for Software Engineering*. Addison Wesley, 1995.

[11] 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 Trans. Software Eng.*, 28 (8):721–734, 2002.

[12] MCM Experimentation, http://www.cs.ust.hk/~cssam/MCM. (Last accessed, Feb 1, 2006.)

[13] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988.

[14] H. Mili, F. Mili, and A. Mili, "Reusing Software: Issues and Research Directions", *IEEE Trans. Software Eng.*, 21(6):528–562, 1995.

[15] 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.

[16] T.H. Ng, and S.C. Cheung, "Proactive Views on Concrete Aspects: A Pattern Documentation approach for Software Evolution", *Proc. 27th Int'l Conf. Computer Software and Applications* (COMPSAC 2003), IEEE Computer Society Press, Dallas, Texas, USA, Nov. 2003, 242–247.

[17] N. Noda, and T. Kishi, "Implementing Design Patterns Using Advanced Separation of Concerns", position paper, *OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, Florida, USA, October 2001, http://www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/submissions/24-noda.PDF. (Last accessed, Feb 1, 2006.)

[18] 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 Trans. Software Eng.*, 27(12):1134–1144, 2001.

[19] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[20] N. Soundarajan, and J.O. Hallstrom, "Responsibilities and Rewards: Specifying Design Patterns", *Proc. 26th Int'l Conf. Software Eng.* (ICSE 2004), IEEE Computer Society Press, Edinburgh, United Kingdom, May 2004, 666–675.

[21] The UML 2.0 Specification, http://www.uml.org/. (Last accessed, Feb 1, 2006.)