Proactive Views on Concrete Aspects: A Pattern Documentation Approach for Software Evolution

T.H. Ng and S.C. Cheung
Department of Computer Science
Hong Kong University of Science and Technology
{cssam, scc}@cs.ust.hk

Abstract

A design pattern provides a structure to manage a design aspect by allowing the aspect to be changed without software redesign. For example, the *Command* pattern allows a software design to be easily reconfigured to replace a receiver of a command request or issue a new command request. To prepare for everlasting demands in diverse requirement changes, multiple design patterns are typically deployed to manage various design aspects of a software program. However, this can complicate the underlying program structure, resulting in difficulties of succeeding program revisions to preserve effective management of aspects. In particular, when design aspects cut across each other, realizing changes in one aspect likely revokes the management of other aspects. To address this problem, we propose a documentation approach to govern program revisions. In the approach, a design aspect is associated with a set of *proactive views*. Each view specifies how an implementation of the aspect is organized so that other aspects can be managed at the same time. The views capture the class relationships that must be kept to preserve effective management of aspects. The approach is illustrated using a pattern-based hotel management system.

Keywords: Design patterns; crosscutting aspects; software evolution; change management

1. Introduction

Software programs keep evolving to meet everlasting requests for revised or additional capabilities. Design patterns have been appreciated as useful solutions to manage software evolutions under anticipated changes [1] [5][7][12][16]. A pattern provides a structure of participants that can be implemented by classes [7][11]. It identifies the responsibilities of each participant to manage a design aspect in a program so that the aspect can be changed without redesign. Figure 1 shows the structure of the *Command*[1] pattern as a UML [13] class diagram. The pattern allows a software design to be easily reconfigured to replace a receiver of a command request or issue a new command request.

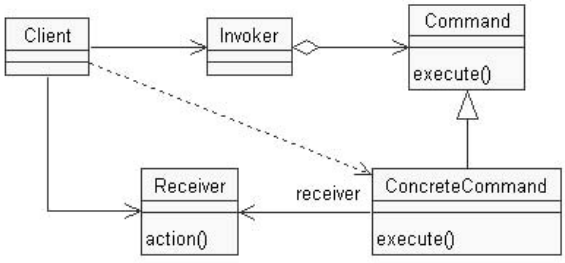

Figure 1: The structure of the *Command* pattern [7]

A *design aspect* represents a specific program property subject to a set of possible settings. When this property varies, its effect may cascade through the entire program. Examples of design aspects are program functionality, user interface appearance, language support, performance strategy, and security policy. The management of a design aspect is generally achieved by using dynamic binding and delegation as indirection techniques to decouple its associated concrete aspects from the other implementations. A *concrete aspect* refers to an implementation of a specific setting of a design aspect. Multiple concrete aspects implementing the same design aspect may co-exist in the program. For example, the functionality design aspect of a program may be implemented by a set of concrete aspects each of which implements a specific functionality of the program.

A *pattern deployment* represents a utilization of the structure of the pattern to manage a design aspect in a program. A deployment is *effective* when each of its associated concrete aspects is encapsulated by a class implementing a particular participant of the pattern. Such

[1] Unless otherwise noted, the names of specific patterns refer to those presented in Gamma et al.'s Design Patterns catalog [7].

a class is referred to as a *hotspot* class, where a hotspot represents an implementation that is subject to change [16]. Appendix A shows the participant to be implemented by hotspot classes for each pattern presented by Gamma et al [7]. The implication of an effective pattern deployment is that an associated concrete aspect can be substituted by simply replacing its encapsulating hotspot class. For example, the *Command* pattern can be deployed to manage the functionality design aspect of a program. Hotspot classes encapsulating the associated concrete aspects must implement the *Concrete Command* participant of the pattern.

To prepare for everlasting demands in diverse requirement changes, multiple patterns are typically deployed to manage different design aspects. Additional levels of indirection are thus introduced in the program [7]. This complicates the underlying program structure [1][4], leading to difficulties in preserving the effectiveness of each pattern deployment in succeeding program revisions.

One major challenge of the problem lies in the fact that a concrete aspect may not entirely reside in its encapsulating hotspot class. More often, the concrete aspect is partitioned into pieces, which reside in different classes. For example, in a typical *Command* pattern deployment, the associated concrete aspects are implementations of the functionality design aspect of the program. These implementations do not entirely reside in their encapsulating hotspot classes. Rather, these classes delegate their functionalities to classes that implement the *Receiver* participant, which may in turn delegate their functionalities to classes that do not implement any participant of the *Command* pattern.

The partitioning of a concrete aspect into pieces is likely required by pattern deployments to manage crosscutting [10] design aspects, where changes in one design aspect affect concrete aspects of other design aspects. For example, suppose that the *Command* and the *Abstract Factory* patterns are deployed to manage the program functionality and user interface appearance design aspects respectively. Each function of a program is associated with a specific user interface. As such, a change in the appearance aspect cascades through all functions. On the other hand, the support of a new function requires addition of a set of user interfaces each of which pertains to a particular supported appearance. Therefore, when realizing changes in a design aspect, considerations must be made to avoid revoking the management of the crosscutting aspects. This consideration can be non-trivial because the associated concrete aspect pieces are possibly scattered throughout the program.

This paper proposes a documentation approach to prepare for changes in design aspects managed by pattern deployments in a program. When the program is revised to realize these changes, the documentation helps identify the followup activities that should be carried out in order to preserve the effectiveness of each pattern deployment.

The organization of this paper is as follows: Section 2 investigates the related work. Section 3 describes our documentation approach in detail. Section 4 discusses the characteristics and evaluations of the approach, followed by a conclusion in Section 5.

2. Related Work

2.1 Case Studies on the Usefulness of Design Patterns in Program Maintenance

Prechelt et al [15] conducted a controlled experiment to study if the deployment of design patterns can facilitate program maintenance. In most cases, positive results were obtained, which generally confirmed the usefulness of design patterns on software evolvability. However, negative results were found in a few cases where the design solutions without deploying patterns were less error-prone or subject to shorter maintenance time. Bieman et al [2] conducted an industrial case study to explore the relationships between design patterns, design structures and program changes. No concrete evidence was found to suggest that design patterns ease adaptability. However, the two studies did not mention which design aspects are managed by the patterns deployed in the experiments. There were also no comments on whether or not the effectiveness of the pattern deployments was preserved across program revisions. As such, it is unclear if the experimental results are directly applicable to the situations involving multiple crosscutting design aspects.

Yet, we believe that deploying design patterns could facilitate software evolution in principle. This is because each pattern allows an aspect to be changed without redesign. In order to benefit from deploying patterns in practice, one should keep precise documentation of patterns deployed. The usefulness of the pattern documentation was studied by Prechelt et al [14]. The results obtained from two controlled experiments indicated that pattern documentation governs revisions of programs with complicated program structures. We believe that pattern documentation is required to preserve the effective management of design aspects in each program revision. This is particularly the case when crosscutting design aspects are managed by pattern deployments.

2.2 Documenting Patterns by Pattern Structure

A pattern deployment can be documented as a mapping from the participants in a pattern structure to their implementing classes [7][12]. There are at least two advantages with this documentation. First, each mapping identifies how the pattern structure is utilized in the program. This reminds software developers of the responsibilities of the mapped classes in a pattern deployment and the rationale of the relationships between

these classes. Second, the mapping identifies the hotspot classes for the design aspect being managed by the pattern deployment. This allows efficient location of the hotspot classes to realize changes in the aspect.

However, there are limitations with this documentation approach. Each mapping may not encompass all the classes that implement the associated design aspect. It also does not indicate if changes in the design aspect affect the management of other design aspects. As a result, while software tools can be applied using this documentation to check the consistency between the mappings and the source code, this does not statically check the effectiveness of each pattern deployment in the program. To statically check the effectiveness, the residence of the associated concrete aspect pieces must be firstly identified. Then, each piece is checked if all its accesses must be through its encapsulating hotspot class. In summary, mapping classes to pattern structures is inadequate to preserve the effectiveness of each pattern deployment.

2.3 Documenting Patterns by Unit Model

Eide et al [6] presented a complementary realization of design patterns, in which many pattern participants correspond to statically instantiated and connected components. The methodology separates the static parts of the software design from the dynamic parts of the system behavior. Using this methodology, pattern documentation is expressed in terms of the unit model of components. The unit model makes it possible for a software architect to design a system from components, describe local and global relationship between components, and reuse components both within and across system designs. With the unit model, these architectural constraints on component compositions can be verified by a constraint checker. Also, this methodology simplifies pattern structures before utilizing them by transforming all dynamic bindings using inheritance hierarchies to delegations. As a result, the documentation purely specifies static configurations of the software program, making the system more amenable to predict run-time behavior. Nevertheless, the documentation does not address crosscutting relationships between design aspects. Thus, it does not manifest the information to preserve pattern deployment effectiveness. Worse, this methodology is only applied to static systems and to static aspects of dynamic systems. In the context of our discussions, it is not applicable to software programs that support interchange of concrete aspects dynamically and programs that are anticipated to support this. To support this, the programs must be re-implemented.

3. Documenting Patterns by Concrete aspects

3.1 Illustrating Example

We firstly describe the design of an example based on a simplified hotel management system (HMS) that supports room reservation. Through the system, a hotel receptionist may perform check-in and checkout operations for guests in separate user interfaces with different appearances dynamically. Let us suppose the developers of the HMS have anticipated two future adaptations: (a) addition of new functionalities, and (b) addition of new appearances for the user interfaces. These two adaptations can be interpreted as changes in two crosscutting design aspects, namely *Command* and *Appearance*. The system design model [3] is shown in Figure 2. Table 1 describes the patterns deployed during the design to cater for the two anticipated adaptations. The HMS has been successfully implemented in Java [8] based on the design model. The program implementation is available at [9].

Let us consider adding an enquiry function to the HMS to support searching for occupants in a hotel. The change can be realized by incorporating a new hotspot class, named *Search*, for the *Command* design aspect into the HMS. The *Search* class can be incorporated as a subclass of the *Command* class in Figure 2. In fact, the implementation of the new function, including the user interface of the function, can reside entirely in the *Search* class. While this is a possible implementation strategy to support a new function in the HMS, it revokes the effective management of the *Appearance* aspect, which requires all implementations of the aspect to be encapsulated by a descendent of the *Skin* class. Nevertheless, the implementation strategy can still lead to a functionally correct HMS system. This is because there is no need to access the *Search* through a subclass of the *Skin* as far as the enquiry function is concerned. However, the implementation strategy makes both the *Abstract Factory* pattern deployment and the *State* pattern deployment for appearances in Table 1 no longer effective. In other words, the two pattern deployments no longer provide the capability to change the design aspect of appearance by substituting only the hotspot class *ListSkin*.

The above scenario raises an interesting question of how to preserve the effectiveness of existing pattern deployments in program revisions. In general, one can assume that the development team remembers how to preserve effectiveness of all pattern deployments in a program if it is revised shortly after delivery. This assumption, however, does not last as human memories likely fade away with the passage of time. In addition, the program revisions, which preserve the effectiveness of various pattern deployments in each program revision, may no longer be followed with the departure of the original development team. Thus, pattern documentation is important to the deployment of design patterns. It is a key to govern the program revisions proactively.

Deployed Patterns	Purpose	Managed Design Aspect	Hotspot Classes
1. **State (for appearances)**	Avoid coupling the UI class from its appearance.	Appearance	ListSkin
2. **State (for commands)**	Avoid coupling the UI class from the active user interface for a particular command.	Command	CheckIn, CheckOut
3. **Abstract Factory**	Avoid particular commands adhere to particular appearances so that commands are portable across different appearances.	Appearance	ListSkin
4. **Command**	Allow the UI class to issue a command request without knowing anything about the receiver of the request.	Command	CheckIn, CheckOut

Table 1: Description of pattern deployments in the hotel management system

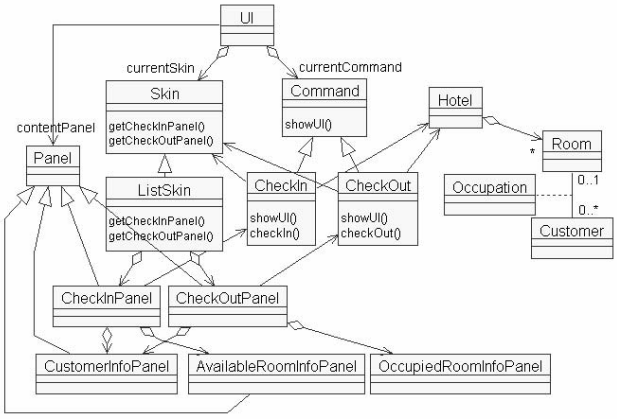

Figure 2: System design of the hotel management system

3.2 Documentation Approach

The basis of our approach is to document each concrete aspect encapsulated by a hotspot class using a proactive view. A *proactive view* is a model based on a projection of a system design model of a program on a concrete aspect. It comprises the encapsulating hotspot class of the concrete aspect and followers of the class. The goal of this documentation is to proactively capture class relationships that must be kept after program revisions under anticipated changes. This imposes necessary conditions to preserve the effectiveness.

Definition 1 (Follower): *A follower of a hotspot class is an implementation piece of the encapsulated concrete aspect with this piece residing outside the hotspot class.*

By definition, since a hotspot class encapsulates a concrete aspect, it must also encapsulate all its followers. This implies an access control on a follower; only the class it follows or the followers of the class can reference it by name. This also implies that when the class is removed from the program, all its followers will not be able to be accessed in the program. Therefore, they become redundant and can be removed accordingly.

A follower may follow more than one class. In this case, it will be removed if one of the classes it follows is removed. There requires a further access control; only one of the classes it follows or an implementation that is a follower of all these classes can access it by name. This is to avoid dangling references as it is removed due to the removed of one of its followers.

Definition 2 (Mandatory and Optional Followers): *A mandatory follower of a hotspot class is a follower of the class and implements a pattern participant. A follower not a mandatory follower is an optional follower.*

If a hotspot class for a design aspect has a mandatory follower, there are two possibilities. First, the follower is a participant of the pattern deployment that manages the design aspect. The second possibility is that the design aspect cuts across other design aspects. Furthermore, the follower participates in pattern deployments to manage these other aspects.

The rationale of this classification is that a mandatory follower is required to follow pattern structures and preserve the effectiveness of each pattern deployment. This implies that it cannot be removed unless the class it follows is removed. In other words, it must be kept when the class it follows is revised.

On the other hand, as specified in the associated pattern structures, a hotspot class and its followers may be required to be subclasses of a certain class or to

implement a certain interface. This requirement can also be documented in the associated proactive view. As such, proactive views serve two major functions.

Firstly, a proactive view explains how the pieces of the associated concrete aspect are scattered through the program. Figure 3 shows a proactive view on the concrete aspect encapsulated by the *ListSkin* class in the HMS. The view specifies the organization of the appearance implementation that is encapsulated by the *ListSkin* class. In the view, the *CheckInPanel* and *CheckOutPanel* classes are mandatory followers. This is because they implement the *Concrete Product* participant of the *Abstract Factory* pattern. This means that the two classes cannot be removed as long as the *ListSkin* class is not removed. This is to follow the structure of the pattern. Note that in the proactive views as presented in this paper, we stereotype mandatory followers as simply followers.

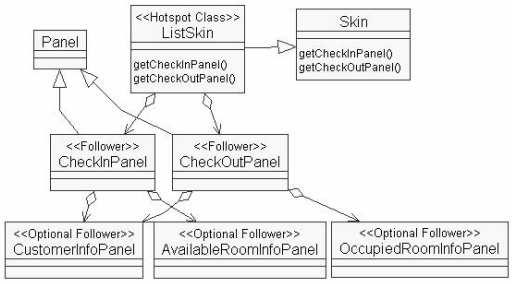

Figure 3: Proactive view on the concrete aspect encapsulated by *ListSkin*

Secondly, a proactive view specifies how the associated concrete aspect may be organized so that other design aspects can be managed simultaneously. Figure 4 shows a proactive view on the concrete aspect encapsulated by the *CheckIn* class in the HMS. This view specifies the organization of the check-in checkout functionality of the HMS. Since the functionality associates with a user interface, the concrete aspect of the functionality design aspect must crosscut concrete aspect of the appearance design aspect. The information of this crosscutting is captured as mandatory followers in the view.

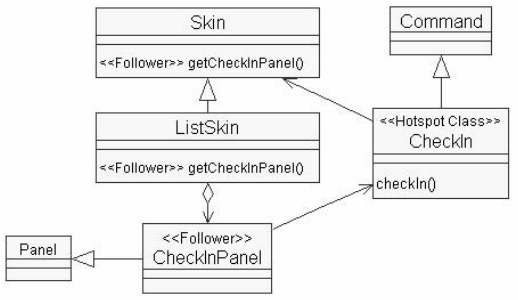

Figure 4: Proactive view on the concrete aspect encapsulated by *CheckIn*

4. Discussions

4.1 Characteristics

This paper presents documenting patterns by concrete aspect. This documentation approach has the following benefits that cannot be found in conventional pattern documentation approaches:

1. A proactive view specifies the access control constraint on the followers of the corresponding hotspot class. This enables statically checking the effectiveness of the associated pattern deployment.

2. A proactive view specifies how the encapsulating concrete aspect is organized so that other aspects can be managed at the same time. This captures the class relationships that must be kept to preserve effective management of aspects. This also serves as a reference organization for new concrete aspects of the design aspect, facilitating both experts and non-experts of design patterns to realize the changes.

The overall approach proposes a pattern documentation approach for software evolution, aiming at preserving effectiveness of each pattern deployment after realizing anticipated changes. It is applicable when design patterns are deployed in a software design. It is particularly useful when pattern deployments deeply complicate the design and design aspects cut across each other.

4.2 Evaluation

Using the proposed approach in software development, the associated software program is attached with a time series set of proactive views. The cardinality of the set is the number of concrete aspects for each design aspect managed by pattern deployments. This number may not be small as multiple design patterns are deployed to manage various aspects. During software maintenance, the views keep being revised and the number keeps increasing accordingly. Indeed, the approach increases workload in all phases of software development. The tradeoff is that the followup information for preserving effectiveness of pattern deployments is manifested for software evolution.

The presentation in this paper so far only discusses the realization of changes in design aspects managed by pattern deployments. As for unanticipated changes, or changes in unmanaged aspects, new followers of hotspot classes may be introduced unexpectedly. Still, the access control constraints specified in the documented proactive views are valid. Proactive views still specify necessary class relationships that must be kept to preserve effective management of aspects.

This paper assumes that the same set of patterns is deployed after program revisions. In fact, more patterns may be deployed or deployed patterns may be switched to

others. Future research will be conducted in these areas to improve software evolvability as the program evolves.

5. Conclusion

The deployment of multiple patterns to manage various design aspects complicates the underlying program structure. This results in difficulties in program revisions to preserve effectiveness of each pattern deployment. To address this problem, this paper proposes a documentation approach to govern the program revisions when realizing changes in design aspects. Using the approach, each concrete aspect encapsulated by a hotspot class is associated with a set of proactive views. These views capture necessary class relationships to preserve the effectiveness of each pattern deployment for program revisions under anticipated changes. Future research will focus on improving software evolvability in response to occurrences of unanticipated changes.

6. Acknowledgment

This work has been supported by the Hong Kong Research Grant Council with an Earmarked Research Grant (HKUST6187/02E).

Appendix A

Patterns	Participant to be Realized as Hotspot Classes
Abstract Factory	Concrete Factory
Builder	Concrete Builder
Factory Method	Concrete Creator
Prototype	Concrete Prototype
Singleton	Singleton
Adapter	Adaptor
Builder	Refined Abstraction
Composite	Composite, Component
Decorator	Concrete Decorator, Concrete Component
Façade	Façade
Flyweight	Flyweight Factory
Proxy	Proxy
Chain of Responsibility	Concrete Handler
Command	Concrete Command
Interpreter	Nonterminal Expression, Terminal Expression
Iterator	Concrete Aggregate
Mediator	Concrete Mediator
Memento	Memento
Observer	Concrete Observer
State	Concrete State
Strategy	Concrete Strategy
Template Method	Concrete Class
Visitor	Concrete Visitor

Table 2: Participant to be realized as hotspot classes for each pattern

7. References

[1] E. Agerbo and A. Cornils. How to Preserve the Benefits of Design Patterns. *Proceedings of Conference on Object-oriented Programming, Systems, Languages, and Applications*, Vancouver, 1998, pp. 134-143.

[2] J.M. Bieman, D. Jain and H.J. Yang. OO Design Patterns, Design Structure, and Program Changes: An Industrial Case Study. *Proceedings International Conference on Software Maintenance* (ICSM 2001), Florence, November, 2001, pp. 580-589.

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

[4] C. Chambers, B. Harrison and J. Vlissides. A Debate on Language and Tool Support for Design Patterns. *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2000, pp. 277-289.

[5] J.O. Coplien and D.C. Schmidt. *Pattern Languages of Program Design*. Addison-Wesley Publishing Company, 1995.

[6] E. Eide, A. Reid, J. Regehr and J. Lepreau. Static and Dynamic Structure in Design Patterns. *Proceedings of the 24th International Conference on Software Engineering*, Orlando, 2002, pp. 208-218.

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

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

[9] HMS Implementation, http://www.cs.ust.hk/~cssam/hotel.

[10] K. Mens, T. Mens and M. Wermelinger. Supporting software evolution with intentional Software Views. *Proceedings of the International Workshop on Principles of Software Evolution*, Orlando, 2002, pp. 138-142.

[11] J. Noble, R. Biddle and E. Tempero. Metaphor and Metonymy in Object-Oriented Design Patterns. *Proceedings of the 25th Australasian Conference on Computer Science*, Melbourne, 2002, pp. 187-195.

[12] N. Noda and T. Kishi. Implementing Design Patterns Using Advanced Separation of Concerns. *OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems,* Japan, 2001.

[13] Object Management Group. *Unified Modeling Language Specification, Version 1.4.* http://www.omg.org, 2001.

[14] L. Prechelt, M. Philippsen and W.F. Tichy. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. *IEEE Transaction of Software Engineering*, Vol.28, No,.6, June, 2002, pp. 595-606.

[15] L. Prechelt and W.F. Tichy. A Controlled Experiment in MainTenance Comparing Design Patterns to Simpler Solutions. *IEEE Transactions on Software Engineering*, Vol. 27, No. 12, December, 2001, pp. 1134-1144.

[16] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley Publishing Company, 1995.

[17] J. Vlissides. *Pattern Hatching: Design Patterns Applied*. Software Patterns Series. Addison-Wesley, New York, 1998.