Aspect-Oriented Implementation of Concurrent Processing Design Patterns

Shingo Kameyama, Masatoshi Arai, Noriko Matsumoto, Norihiko Yoshida
Graduate School of Science and Engineering
Saitama University
Saitama, Japan
{shingo, arai, noriko, yoshida}@ss.ics.saitama-u.ac.jp

***Abstract*—A variety of design patterns are now widely used in software development as their catalog is a collection of knowledge on design and programming techniques and namely elaborated patterns. However, as each design pattern is described in the forms of texts, charts, and simple code examples, it has some limitations in applicability and formal treatment. One of its reasons is that the design patterns include some crosscutting concerns. To solve this problem, aspect-oriented implementation of the so-called "Gang of Four" (GoF) design patterns, which are cataloged for component reuse has been proposed. In this paper, we propose aspect-oriented implementation of design patterns for concurrent processing, so as to improve and accelerate design and development processes of, for example, network systems, embedded systems, and transaction systems. Our aspect-oriented implementation tailors hierarchical or inclusive relationships among design patterns well which are not found in the patterns for component reuse, but found in the patterns for concurrent processing.**

Keywords-Design patterns; aspects; concurrency

I. Introduction

A variety of design patterns (patterns in short, sometimes, hereafter) [1] are now widely used in software development as their catalog is a collection of knowledge on design and programming techniques and namely elaborated patterns. Their catalog enables novices to refer to experts' knowledge and experiences in software design and development. It accelerates software productivity and improves qualities as well as it helps communication within a development team.

Many researchers have proposed various patterns. However, patterns have sometimes difficulties in formal or systematic treatment. It is because patterns are described in the forms of texts in a natural language, charts, figures, and code fragments and samples, not in any formal description.

To solve this problem, aspect-oriented [2] implementation of patterns for component reuse has been proposed [3][4][5][6][7]. Patterns for component reuse, or sometimes called GoF (Gang of Four) design patterns, were patterns promoting component reuse in object-oriented software [1]. Aspects are a technique to modularize codes which are scattered to several modules [2].

In this paper, we propose aspect-oriented implementation of concurrent processing design patterns [8][9][10]. Different from GoF design patterns, most concurrent processing patterns use other concurrent processing patterns related to them, namely, they are not independent to each other. We implement them by a combination of the implementation of related concurrent processing pattern. This approach can be applied also to aspect-oriented implementation of other design patterns where there are some mutual dependency among them.

This paper is organized as follows: Section 2 gives an overview of design patterns. Section 3 presents an overview and some functions of aspect-orientated implementation. Section 4 introduces an example of aspect-oriented implementation of GoF design patterns. Section 5 and Section 6 explain aspect-oriented implementation of concurrent processing design patterns. Section 7 gives some considerations, and Section 8 contains some concluding remarks.

II. Design Patterns

Design patterns are a catalog of typical solutions for some typical problems in designing and programming in software development. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, so-called the Gang of Four (GoF), introduced 23 design patterns for reuse (so-called the GoF design patterns) [1]. Design patterns have been getting widely accepted as a technique to improve software development processes. Many researchers have proposed various design patterns for concurrent processing, real-time processing, and web applications, etc.

A. Benefits

Design patterns accelerate software productivity and improves qualities in the following regards.

- A workload for programming and verification can be reduced reusing codes in design patterns.
- A development beginner can use it as a guide in which development experts' know-how is accumulated.
- A developer can tell a software design to another developer concisely and precisely by describing the name of a design pattern.

B. Problems

A design pattern has the following problems because its codes are scattered to more than one module.

- A developer must understand the structure of a design pattern to apply, extract some necessary codes from its sample codes, and apply to a program.
- It is difficult to maintain or extend the program with design patterns because codes relating the patterns are scattered across several modules, and a developer must keep all of them in mind correctly.

C. Concurrent Processing Design Patterns

Besides GoF design patterns, concurrent processing design patterns [8][9][10] have been proposed. A concurrent processing design pattern offers a typical solution to the following problems in concurrent processing.

- There are problems, such as race conditions and deadlocks, that do not happen in single threaded processing. A race condition happens when more than one thread read and write a shared resource without mutual exclusion control. A deadlock happens when more than one thread lock more than one object and wait for unlocking with each other.
- Verification is difficult because any problem may or may not occur, but possibly not always.

Different from the GoF design patterns for component reuse, most design patterns for concurrent processing have relations with each other, i.e., they are not independent to each other, and most patterns use other patterns related to them. In other words, there is an hierarchy among them. We explain concurrent processing design patterns below. Figure 1 shows the hierarchical relation among them. An arrow denotes that a design pattern on the head side uses a design pattern on the tail side.

Single Threaded Execution (or Critical Section)
: It ensures safety by exclusive control. Only a single thread can access a thread-unsafe object at a time.

Immutable
: It improves throughput by eliminating changing a state of an object and exclusive control.

Guarded Suspension (or Guarded Waits or Spin Lock)
: It ensures safety by blocking a thread until a state of an object changes if a precondition is not met.

Balking
: It ensures safety and improves responsibility by not executing a processing if a precondition is not met.

Producer-Consumer
: It ensures safety and improves throughput by passing an object indirectly.

Read-Write Lock (or Readers and Writers)
: It ensures safety and improves throughput by allowing concurrent access for read and requiring exclusive access for write.

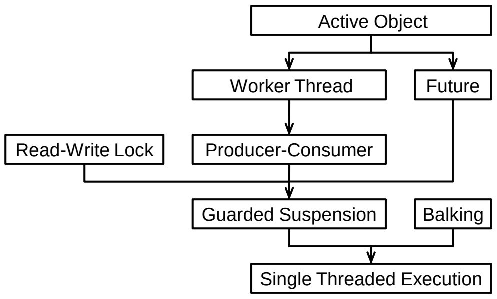

Figure 1. Inclusive relation of concurrent processing design patterns.

Thread-per-Message (or Thread-per-Method)
: It improves responsibility by creating a thread every processing request, and leaving the processing to the thread.

Worker Thread (or Thread Pool)
: It improves responsibility, throughput, and capacity by leaving a requested processing to another thread, and reusing the thread.

Future
: It improves responsibility by enabling a thread to receive a result when it is needed, instead of waiting for a result.

Two-Phase Termination
: It ensures safety by terminating another thread indirectly.

Thread-Specific Storage (or Thread-Specific Data)
: It enables a thread to execute the thread-specific processing by providing the thread-specific storage.

Active Object (or Actor)
: It improves responsibility and enables more than one thread to request a processing to a thread-unsafe object at a time by leaving the processing to a single thread.

III. Aspects

Object-oriented programming is a technique that modularizes a concept and a concern as an object, and improves maintainability and extensibility. However, modularization by objects has a limitation because codes of a crosscutting concern related to more than one module are scattered on them. A crosscutting concern is a functionality of the system whose definition appear in several classes. Examples of crosscutting concerns include logging and caching.

Aspect-oriented programming [2] is a technique that compensates the above-mentioned limit in object-oriented programming and modularizes crosscutting concerns.

A. Overview

In object-oriented programming, a module calls a method in another module to execute. When a module is added or removed, it is necessary to adjust all the method calls related to it in all the other modules. Figure 2 shows a class diagram depicting this scheme. An arrow between methods expresses a processing flow.

In aspect-oriented programming, a method or a code fragment specifies positions in other method definitions in any module where it must be called. This method or code fragment runs when execution reaches the position. The specification of the position is called a pointcut. The code fragment is called an advice. This module composed of pointcuts and advices is called an aspect.

When a module is added or removed, it is not necessary to adjust other modules that call it because there are no explicit method calls related to it. Figure 3 shows a class diagram depicting this scheme. A pointcut and an advice are described at the bottom part of boxes which represent aspects. A dashed line expresses a specification by a pointcut.

B. AspectJ

AspectJ is an aspect-oriented language extension adding the aspect-oriented features to the object-oriented language Java. Java and AspectJ are used in a related study [3] and this study. We explain some functions of AspectJ here.

Pointcut
: A position that can be specified by a pointcut is limited to a position where a method is called and executed, and where a field value is retrieved or assigned.

Advice
: An advice includes a before advice, an after advice, and an around advice. A before advice is executed just before processing of the position specified by a pointcut. An after advice is executed just after it. An around advice is executed instead of it. In an around advice, *proceed* which means execution of the original method can be specified.

Aspect
: Like a class, an aspect can include fields and methods, and can be defined as an abstract aspect or a concrete aspect. An aspect can also inherit another aspect.

IV. RELATED WORKS

A design pattern includes several classes in general, therefore a pattern itself is a crosscutting concern. Aspects are expected to enable modularization of the design pattern as a crosscutting concern. Aspect-oriented implementation of the GoF design patterns has been proposed in related studies [3].

In this section, we explain aspect-oriented implementation of the *observer* pattern which is one of the GoF design patterns as an example of the related study [3]. The *Observer* is to execute a method whenever a state of another object changes.

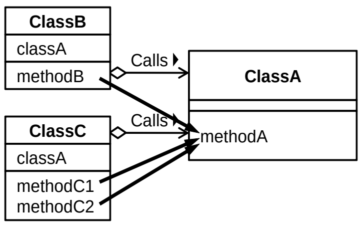

Figure 2. Modularization using objects.

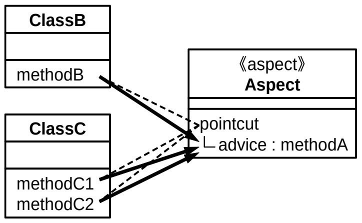

Figure 3. Modularization using aspects.

A. Aspect-Oriented Implementation of Observer

In object-oriented implementation, the *observer* is implemented as follows: a method *notifyObservers* is called just after any change of a state, and calls linked methods. The following codes of *observer* are defined in more than one module. Figure 4 shows a class diagram of this scheme.

- *NotifyObservers* and method calls to execute it.
- A field and methods to manage linked objects.
- A super-class of linked objects.

In aspect-oriented implementation, *observer* is implemented as follows: the code to execute is defined as an after advice instead of *notifyObservers*, and the codes of *observer* is modularized in a single aspect. Figure 5 shows a class diagram of this scheme. *Observer* is actually implemented as an abstract aspect and a concrete aspect. The abstract aspect defines an advice which does not depend on a target program where a design pattern is applied. The concrete aspect defines a pointcut which depends on it. This improves reusability of the abstract aspect.

B. Benefits

There are the following benefits because codes of a design pattern are modularized.

- A developer can apply a design pattern to the position in any target system specified by a pointcut even if he or she does not understand its structure.
- A developer need to update only one module regarding the design pattern when the target system to which the pattern is applied is updated.
- The design pattern is defined separately from the target system, therefore it is easier to maintain and reuse the system.

V. ASPECT-ORIENTED IMPLEMENTATION OF READ-WRITE LOCK

We implemented all the concurrent processing design pattern mentioned earlier as an abstract aspect and a concrete aspect like *observer*, and confirmed that this implementation works correctly.

In this section, we explain aspect-oriented implementation of *read-write lock* as an example of our study. A set of locking

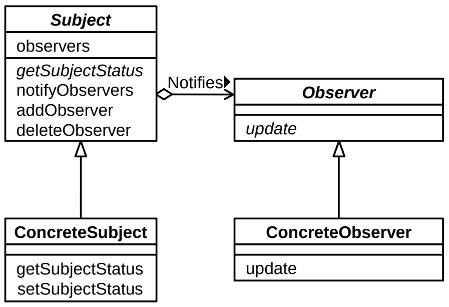

Figure 4. *Observer* implemented using objects.

and unlocking is a typical example of crosscutting concern, and they are to be done before and after read and write in *read-write lock*.

A. Read-Write Lock Defined Using Objects

In object-oriented implementation, *read-write lock* is implemented by testing a precondition and locking just before read and write, and unlocking and notifying a change just after reading and writing under exclusive control. Figure 6 shows a class diagram of this scheme.

Locking and unlocking are done by counting reading and writing threads, and the counters are used for testing a precondition. The precondition for read is that there is no writing thread. The precondition for write is that there is no reading and writing thread. A thread waits until the state of the counter changes if the precondition is not met. Exclusive control is done by an instance lock that is a feature of Java. The instance lock is also necessary to execute *wait* and *notifyAll*.

B. Read-Write Lock Defined Using Aspects

Guarded suspension is used for testing a precondition and notifying a change of the state in *read-write lock*, and *single threaded execution* is used for exclusive control in *guarded suspension*. This inclusive, or hierarchical relation is shown in Figure 1.

In aspect-oriented implementation, we implement the *read-write lock* including aspect implementations of *single threaded execution* and *guarded suspension*. Figure 7 shows a class diagram of *read-write lock*.

It is necessary to apply the aspect of *single threaded execution* before the aspect of *guarded suspension*. The first reason is that testing a precondition and notifying a change in state are done under exclusive control. The second reason is that *wait* and *notifyAll* are used. An abstract aspect of *guarded suspension* defines a precedence of concrete aspects by adding + after names of the abstract aspects because if a name of a concrete aspect is used, the abstract aspect need to be updated when adding, deleting, or changing it.

In object-oriented implementation, because a class manages a lock and counters, an instance of the class is created for every object to be read and written. In aspect-oriented implementation, because the aspect manages the lock and counters. A test of preconditions for reading and writing is defined in two concrete aspects, and the aspects are defined as privileged to access to the counter which is a private field in another aspect.

VI. Aspect-Oriented Implementation of Other Concurrent Processing Design Patterns

In this section, we present brief summaries of aspect-oriented implementation of the other concurrent processing design patterns. If a design pattern uses other concurrent processing design patterns, we implemented the design pattern by including the aspects for them as shown in the *read-write lock*.

We are not successful yet in implementing *thread-specific storage* in aspects. The *Thread-specific storage* is a design pattern to re-implement an object as an object with the same API, and such a major structural change is difficult to implement using aspects.

Immutable

Immutable cannot be implemented by an advice because the pattern does not execute any code. Instead, We implemented a checking feature whether a field of a target object is assigned from outside the object or not.

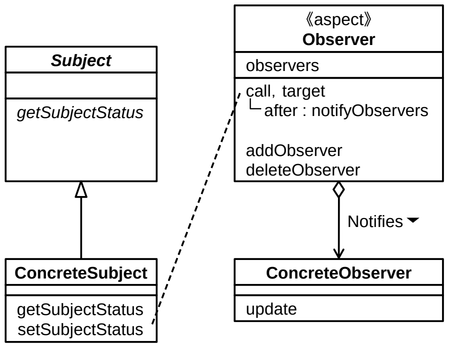

Figure 5. *Observer* implemented using aspect.

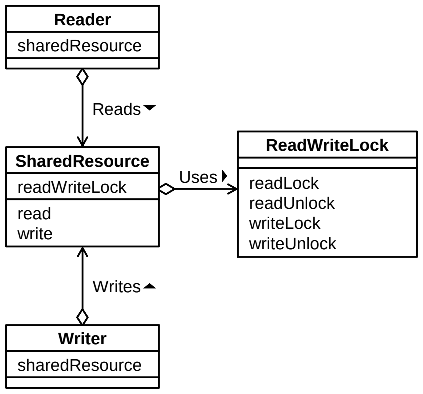

Figure 6. *Read-Write Lock* using objects.

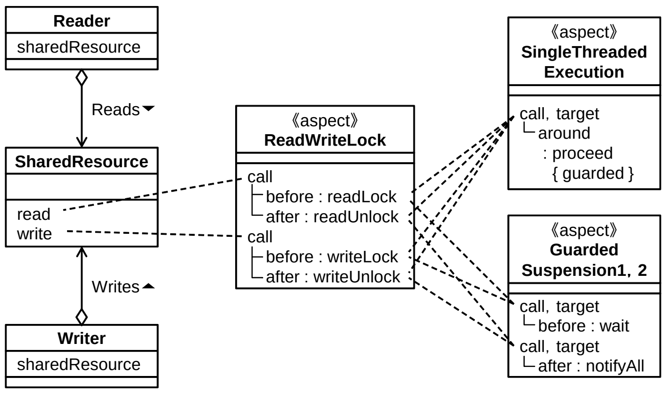

Figure 7. *Read-Write Lock* using aspects.

Balking
: The pattern needs a feature to terminate a method execution, however an advice has no feature to do it. Therefore, we implemented *balking* using an around advice. *Proceed* (i.e. the original code in the target object) is executed if a precondition is met. The around advice is terminated if a precondition is not met.

Producer-Consumer
: Codes for a producer and a consumer must specify codes for send and receive. However, it is difficult for a pointcut to specify an advice because an advice has no name. In our implementation of *producer-consumer*, an advice executes a method to send or receive an object. An pointcut specifies the name of the methods.

Thread-per-Message
: When applying *thread-per-message* to a program afterward, in object-oriented implementation, a developer needs to modify the program so that another thread may execute a requested processing. In aspect-oriented implementation, a developer need not modify the program. An around advice handles the processing, and another thread executes the original code.

Worker Thread
: In aspect-oriented implementation, an around advice is executed instead of a requested processing, and an abstract aspect defines an inner class that executes the original code using the *proceed* feature.

Future
: In aspects implemented *thread-per-message* and *worker thread*, an advice returns the same instance as a return value of a requested processing. The instance and the processing result are related using a map feature.

Two-Phase Termination
: In aspect-oriented implementation, a thread terminates itself executing *stop* at a safe position. It is because an advice cannot terminate the execution.

Active Object
: When adding a method to a thread-unsafe object, in object-oriented implementation, a developer must implement a new task to execute the method, and add a method to create the task. In aspect-oriented implementation, a developer need only to implement a new concrete aspect of *worker thread*.

In any implementation summarized above, an abstract aspect only defines the abstract structure, and a concrete aspect defines the number and the type of an object to be created. The types of an argument and a return value of the advice are not specific but general *Objects*. These disciplines are to improve reusability of the abstract aspects.

VII. Consideration

We express the inclusive relations among concurrent processing design patterns as a combination of aspects. Regarding this, we had to do some refactoring on aspect implementations of the patterns.

For example, when including an aspect for *single threaded execution* in an aspect for *guarded suspension*, first implementation did exclusion control using an instance lock of the aspect for *single threaded execution*, and the aspect for *guarded suspension* could not execute *wait* and *notify*. Therefore, we modified the aspect for *single threaded execution* so that an instance lock of an object can be used.

However, when including the aspects in an aspect of *read-write lock*, a change in state could not be notified between concrete aspects because *wait* and *notify* are instance methods of the aspect of *guarded suspension*. Therefore, we modified the aspect of *guarded suspension* moreover so that an instance method of a target object can be used.

This approach can be applied also to other aspect-oriented implementations where a design pattern includes other design patterns, as well as concurrent processing design patterns.

VIII. Conclusion and Future Work

In this paper, we propose aspect-oriented implementation of concurrent processing design patterns, and if a design pattern uses other concurrent processing design patterns, we implemented the design pattern by including aspects implementing them.

We are still at the starting point in this research, and there is still much to do. Below are some future research directions:

- Refinement of aspect-oriented implementation of concurrent processing design patterns.
- Methodology of aspect-oriented implementation and refactoring in inclusion of aspects.
- Categorization of concurrent processing design patterns from the point of aspect-oriented implementation.

References

[1] E, Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns, Addison-Wesley, 1994.

[2] R. Laddad, AspectJ in Action, Manning, 2003.

[3] J. Hannemann and G. Kiczales, “Design Pattern Implementation in Java and AspectJ”, Proc. ACM OOPSLA, 2002, pp. 161–173.

[4] M.L. Bernardi and G.A. Di Lucca, “Improving Design Pattern Quality Using Aspect Orientation”, Proc. IEEE STEP, 2005, pp. 206–218.

[5] A. Garcia, C. Sant’Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa, “Modularizing Design Patterns with Aspects: a Quantitative Study”, Proc. ACM AOSD, 2005, pp. 3–14.

[6] M. Bynens and W. Joosen, “Towards a Pattern Language for Aspect-Based Design”, Proc. ACM PLATE ’09, 2009, pp. 13–15.

[7] Z. Vaira and A. Caplinskas, “Case Study Towards Implementation of Pure Aspect-Oriented Factory Method Design Pattern”, Proc. IARIA PATTERNS ’11, 2011, pp. 102–107.

[8] D. Lea, Concurrent Programming in Java, Addison-Wesley, 1999.

[9] A. Holub, Taming Java Threads, Apress, 2000.

[10] M. Grand, Patterns in Java, Volume 1, Wiley, 2002.