Implementing Design Patterns in AspectJ and JavaMOP *

Gergely Kovásznai
Department of Information Technology
Eszterházy Károly College, Eger, Hungary
kovasz@aries.ektf.hu

December 20, 2010

Abstract

In this report, we primarily investigate what are the strengths of aspect-oriented programming in implementing design patterns. Furthermore, we propose a higher-level specification layer by employing monitoring-oriented programming (MOP). Since we develop a generic Java framework for design pattern specification, we use AspectJ and JavaMOP . In this report, we implement the Observer design pattern, for the sake of demonstrating the usefulness of those tools. Furthermore, we show a case study of using our generic framework.

Contents

1 Introduction 2

2 Aspect-Oriented Programming and AspectJ 2
2.1 Installing and Using AspectJ 3

3 Monitoring-Oriented Programming and JavaMOP 4
3.1 Installing and Using JavaMOP 6

4 Our Generic Framework 7

5 The Observer Pattern 7
5.1 Interfaces 8
5.2 Abstract Aspect 9
5.2.1 Auxiliary Fields and Methods 9
5.2.2 Abstract Auxiliary Methods 9

*Supported by the Austrian-Hungarian Intergovernmental S&T Cooperation Program (TéT) under the contract AT-25/2008

5.2.3 Pointcuts . . . 10
5.2.4 Advice Actions . . . 11
5.2.5 Error Handlers . . . 12
5.3 JavaMOP Monitor . . . 13
5.3.1 Events . . . 13
5.3.2 Finite State Machine . . . 14

6 Conclusion 17

A Case Study for the Observer Framework 18
A.1 Classes . . . 18
A.2 Aspect . . . 19
A.2.1 Classes' Parents . . . 20
A.2.2 Implementing Abstract Auxiliary Methods . . . 20
A.2.3 Advice . . . 21
A.3 Main Method . . . 21

1 Introduction

Aspect-oriented programming seems an advantageous paradigm when one would like to specify constraints on classes, and to make a software framework able to check at runtime if those constraints are not violated. On the top of that, another paradigm, monitoring-oriented programming is about handling the violation of constraints.

In this report, we investigate these two paradigms in connection with design pattern specifications. We are about implementing generic design pattern frameworks in Java, hence we investigate aspect-oriented and monitoring-oriented programming tools for Java. In Section 2 and 3, the tools AspectJ and JavaMOP are introduced.

In Section 4, we propose the general idea of our generic design pattern framework. In Section 5, we demonstrate how to build such a framework for a given pattern, namely for the Observer pattern. In Appendix A, a case study of using our Observer pattern framework is introduced.

2 Aspect-Oriented Programming and AspectJ

As it is written in [2], "The basic premise of aspect-oriented programming is to enable developers to express modular *cross-cutting concerns* in their software. (...) A cross-cutting concern is behavior, and often data, that is used across the scope of a piece of software."

For a cross-cutting concern, an *aspect* is another term. In aspect-oriented programming, aspects can be specified in a modular way. For this, three types of constructs are used:

Join points: Specific points within an application of which the nature is dependent on the tools being used. In AspectJ [4], which is an aspect-oriented extension to Java, the following join points are supported, among others:

- join when a method is called
- join when a constructor is invoked
- join during object initialization
- join during static initializer execution
- join when a class' field is referenced/assigned
- etc.

Pointcuts: Pointcuts are for declaring an interest in a joint point. They encapsulate the decision-making logic that is evaluated every time when a joint point is encountered.

Advices: The code that is executed when a join point has been triggered. Together with the advice, it is also specified when it is to be invoked in the relation to the join point.

In Figure 1, it is illustrated what the relationship between the aforementioned constructs is.

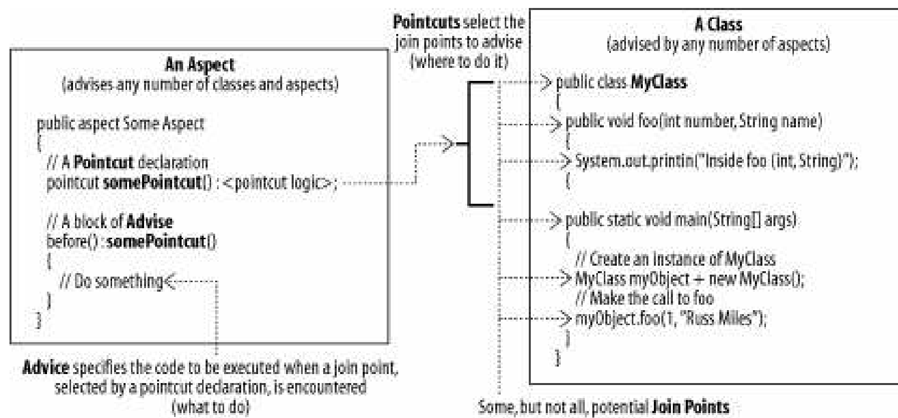

Figure 1: The relationship between join points, pointcuts, and advices.

2.1 Installing and Using AspectJ

Prerequisites. *Java SE Development Kit* (JDK) must be installed. We installed Java SE 6 Update 23. Similarly, *Eclipse IDE for Java developers* must be downloaded and executed. We used Eclipse 3.6.1.

The most convenient way for using AspectJ is to install the *AspectJ Development Tools* (AJDT) [5], which is an Eclipse platform aspect-oriented software development with AspectJ. We used AJDT 2.1.1. for Eclipse 3.6.

To install AJDT, one must select the menu item "Install New Software..." in Eclipse, and then adding the AJDT zip archive as a new repository, as can be seen in Figure 2. We recommend all AJDT tools but its source code installed.

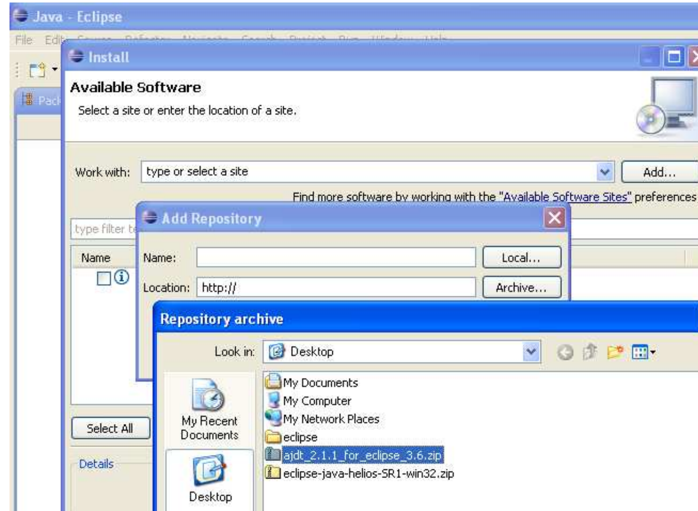

Figure 2: Installing AJDT in Eclipse.

After this, one can create a new AspectJ project, as can be seen in Figure 3. When such a project has been created, one can add a new aspect to it, as can be seen in Figure 4.

3 Monitoring-Oriented Programming and JavaMOP

Monitoring-oriented programming (MOP) is a "programming paradigm built upon runtime verification intuitions and techniques, aiming at supporting reliable software development via monitoring and recovery. MOP takes monitoring of system requirements as a fundamental software development principle, providing an extensible formal framework to combine implementation and specification. The user of MOP specifies desired properties using definable formalisms. Monitoring code is then automatically generated from properties and integrated into the original program; the role of the monitoring code is to verify the runtime behavior of the system against the specified properties. Traditional runtime verification approaches mainly focus on detecting violations. MOP goes one step beyond that by allowing and encouraging the user to provide code to execute when properties are violated or validated, which can be used not only to report

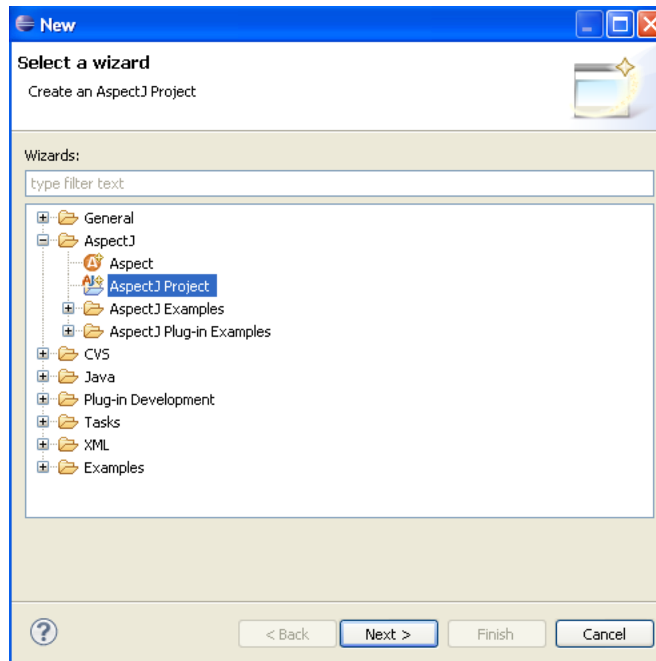

Figure 3: Creating a new AspectJ project in Eclipse.

but especially to recover from errors at runtime." [6]

One of the several layers of the MOP framework contains so-called *logic plugins*, which synthesize monitors in a programming language independent way, e.g., as finite state machines (c.f., Section 5.3.2).

JavaMOP [7] is a MOP development tool for Java. Its specification processor employs AspectJ for monitor integration. "In other words, JavaMOP translates outputs of logic plugins into AspectJ code, which is then merged within the original program by the AspectJ compiler." JavaMOP currently supports 6 logic plugins:

- finite state machines (FSM)
- extended regular expressions (ERE)
- context-free grammars (CFG)
- past time linear temporal logic (PTLTL)
- future time linear temporal logic (FTLTL)
- past time linear temporal logic with calls and returns (PTCaRet)

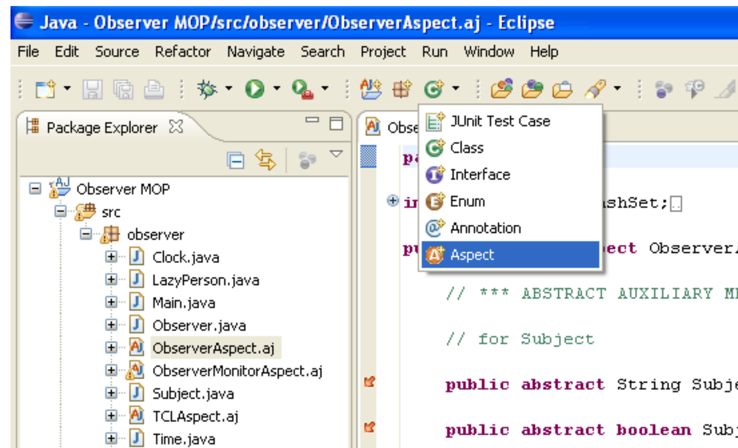

Figure 4: Adding a new aspect to a project in Eclipse.

3.1 Installing and Using JavaMOP

Prerequisites. *AspectJ* [4] must be installed, since JavaMOP intends to use the AspectJ compiler (`ajc.exe`). The installer is a JAR file which one can execute by double-click. We installed AspectJ 1.6.10.

The JavaMOP installer (we used JavaMOP 2.1.2.) requires some modifications on environment variables:

- Add the JDK `bin` folder and the AspectJ `bin` folder to the `PATH` variable.
- Add AspectJ's `lib/aspectjrt.jar` file to the `CLASSPATH` variable.

After installing JavaMOP, it is recommended to add JavaMOP's `bin` folder to the `PATH` variable, since the `javamop` compiler has to be executed from command line. For example, having an `observer.mop` JavaMOP source file, one can compile it by typing "`javamop observer.mop`". The output of `javamop` is an AspectJ source file, which should be added manually to your AspectJ project. Let us note that JavaMOP's `lib/external/commons-collections-3.2.jar` library must be referenced from such projects.

Usually, the `javamop` compiler requires to increase default stack size. Hence, one may need to modify the `javamop` script by using the `-Xss` java switch. For instance, in order to compile the JavaMOP source written in Section 5.3, we used the following `javamop` script:

```
java -cp "c:\javamop2.1\lib\javamop.jar;
c:\javamop2.1\lib\logicrepository.jar;
c:\javamop2.1\lib\plugins\*.jar;
c:\javamop2.1\lib\mysql-connector-java-3.0.9-stable-bin.jar"
-Xss4096k
javamop.Main %*
```

Note that quite a big stack is necessary for even such a small JavaMOP project as the one proposed in Section 5.3.

It is interesting that the JavaMOP installer installs only 3 logic plugins (from all the 6 ones): FSM, ERE, and CFG. The other 3 plugins (e.g., PTLTL) do not seem well-supported, not even on JavaMOP webpage.

4 Our Generic Framework

When implementing a specific design pattern which can be used in a generic way, the following software components are needed to be implemented in advance:

Interfaces: Interfaces that correspond to *roles* in the given pattern. For example, in the Observer pattern (c.f., Section 5), two roles are distinguished: "subject" and "observer".

Abstract aspects: They capture the common requirements to be checked for each concrete application applying the given pattern. They also specify the pointcuts and advice actions for MOP events.

MOP monitors: They capture the higher-level constraints for the given pattern. A MOP code is abstract enough not to depend on concrete applications.

Given such a generic pattern, one can incorporate it in a concrete application, by implementing further components:

Classes: They model the object types related to the application. Let us note that any pattern-specific code can be eliminated from these classes by moving it *into the aspects*.

Aspects: They inherit all requirements from the aforementioned abstract aspects. These concrete aspects are typically for

- assigning the roles (i.e., interfaces) to the classes;
- implementing all the abstract methods of the abstract aspects;
- extending the classes with auxiliary fields or methods, which are accessed by aspects;
- extending the abstract aspects with new pointcuts or advices.

5 The Observer Pattern

The Observer pattern is illustrated in Figure 5. There are two roles in this pattern, "subject" and "observer". Observers that have been enrolled by the subject are to be notified whenever the state of the subject changes.

The framework we are going to propose is based on the one in [3]. However, we have improved the idea by

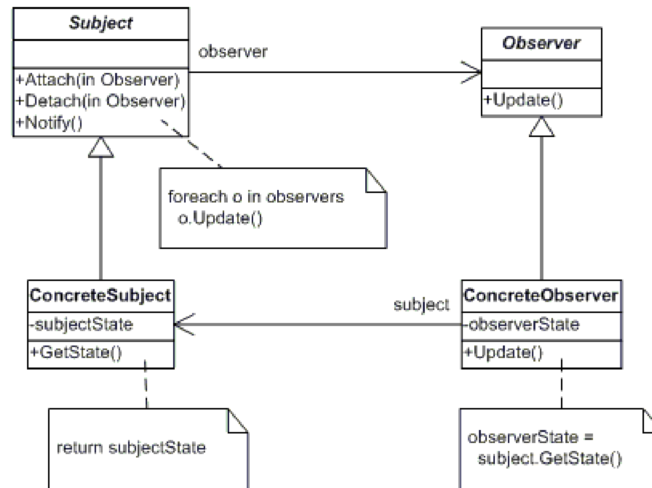

Figure 5: Observer pattern.

- applying Java generics, in order to make the framework type-safe;
- transforming the fields and methods of the abstract aspect into auxiliary fields/methods of the subject or the observers, for the sake of clearer object-oriented model;
- moving all the pattern-specific constraints into the aspects, in order to separate business logic from pattern-specific knowledge completely;
- moving all the possible pointcuts into the abstract aspect, in order to let the concrete aspects inherit them;
- correcting an error.

Let us propose our Observer framework in the subsequent sections.

5.1 Interfaces

Since only two roles can be identified in the Observer pattern, two interfaces are needed, `Subject` and `Observer`.

```
public interface Subject {
}

public interface Observer {
}
```

Note that no field or method is defined in these interfaces. They are really to formalize *roles*, but not behavior.

5.2 Abstract Aspect

In our Observer framework, only one abstract aspect is needed. Its code contains several elements: auxiliary fields and methods, abstract auxiliary methods, pointcuts, advice actions, and error handlers.

5.2.1 Auxiliary Fields and Methods

In the following piece of code, the header of the abstract aspects can be seen, together with auxiliary fields and methods:

```
public abstract aspect ObserverAspect {

  private Set<Observer> Subject.observers;

  public void Subject.attach(Observer o) {
    observers.add(o);
    o.update(this);
  }

  public void Subject.detach(Observer o) {
    observers.remove(o);
  }

  public void Subject.notifyObs() {
    for (Iterator<Observer> e = observers.iterator(); e.hasNext();) {
      e.next().update(this);
    }
  }

  private Set<Observer> Subject.updateCalls = new HashSet<Observer>();

  private String Subject.recordedState;
```

The interface `Subject` is "extended" by three fields and three methods. Of course, as a matter of fact, not the interface itself is extended, but each class which implements this interface (and which is in fact not yet known at this point of the development).

The field `observers` is to enroll the observer objects which are listening to the changes of the subject. Note that the method `notifyObs` is to notify them in the case of a change.

The fields `updateCalls` and `recordedState` play an important role in ensuring the proper functioning of the pattern, as it will be detailed later.

5.2.2 Abstract Auxiliary Methods

Now, let us propose abstract auxiliary methods for `Subject` and `Observer`:

```
public abstract String Subject.getState();
public abstract String Observer.getState();

public abstract boolean Subject.StateModified(String previousState);

public abstract boolean Subject.StateConsistentWith(String observerState);
```

As it is discussed in [3], it is not clear in the Observer pattern when one can say that the state of the subject has changed. Another question is when the state of the subject and the state of one or more observers become *inconsistent* with each other, and hence, when to notify the observers of the change of the subject's state.

To provide the greatest flexibility in our framework, we have to give the opportunity for the developers to implement

- the `Subject.StateModified(String previousState)` method in order to formalize if the current state of the subject (returned by `Subject.getState()`) is considered as *changed* as compared to its previous state;
- the `Subject.StateConsistentWith(String observerState)` method in order to formalize if the current state of the subject is *consistent* with an observer state.

This is why the aforementioned methods are abstract.

Let us note that the state of a subject or an observer is described as a *string*, similarly as in [3]. Of course, a more reasonable solution could be proposed for the same purpose.

One more abstract method has to be specified:

```
public abstract void Observer.update(Subject subject);
```

As it was already shown, this method is called by the `Subject.notifyObs()` method.

5.2.3 Pointcuts

Several pointcuts have to be specified, as follows:

- One needs to capture when a subject has been enrolled, i.e., the constructor of the subject has been executed.

  ```
  protected pointcut subjectEnrollment(Subject subject):
    this(subject) && execution(Subject+.new(..));
  ```

  As can be seen, any constructor of any subclass of `Subject` is referenced by this pointcut.

  Let us note that this pointcut was faultily formalized in [3].

- For capturing the attaching of an observer to a subject, let us specify the following pointcut:

```
protected pointcut attachObserver(Subject subject,
                                  Observer observer):
  target(subject) && call(void Subject.attach(Observer))
  && args(observer);
```

- For capturing the moment when a subject notifies all of its observers:

```
protected pointcut notify(Subject subject):
  target(subject) && call(void Subject.notifyObs());
```

- For capturing the moment when an observer is updated by a subject:

```
protected pointcut update(Subject subject,
                          Observer observer):
  target(observer) && call(void Observer.update(Subject))
  && args(subject);
```

- For capturing the call of any method of the subject:

```
protected pointcut subjectMethod(Subject subject):
  target(subject) && call(public void Subject.*());
```

5.2.4 Advice Actions

By an advice action, we mean a method which captures an advice body. As can be seen in Section 5.3.1, these methods are going to be invoked by JavaMOP events.

- After a subject has been enrolled, its auxiliary fields have to be initiated.

```
void Subject.afterSubjectEnrollment() {
  observers = new HashSet<Observer>();
  recordedState = getState();
}
```

- As it was shown in 5.2.1, an observer gets automatically updated when it is being attached to a subject. The following two advice actions are used to check if that update has been completed.

```
void Subject.beforeAttachObserver(Observer observer) {
  updateCalls.clear();
}

void Subject.afterAttachObserver(Observer observer) {
  if (!updateCalls.contains(observer)) {
    notifyError();
  }
}
```

- Similary, the following two advice actions are used to check if all the observers were updated during the subject was about notifying them. Furthermore, in the `Subject.recordedState`, the current state of the subject gets stored, for further use.

```
void Subject.beforeNotify() {
  updateCalls.clear();

  recordedState = getState();
}

void Subject.afterNotify() {
  if (!updateCalls.containsAll(observers))
    notifyError();
}
```

- The first of the following two advice actions contributes to the check kept by the aforementioned advice action. The second one verifies if the state of the subject is consistent with the state of the observer after its update.

```
void Subject.beforeUpdate(Observer observer) {
  updateCalls.add(observer);
}

void Subject.afterUpdate(Observer observer) {
  if (!StateConsistentWith(observer.getState()))
    updateError();
}
```

- After invoking any methods of the subject, it has to be checked if its observers has been notified.

```
void Subject.afterSubjectMethod() {
  if (StateModified(recordedState))
    subjectMethodError();
}
```

5.2.5 Error Handlers

By an error handler, we mean a method which is invoked in case of a specific error occurring. The main purpose of the error handlers is to detect their invocation by JavaMOP events.

```
void Subject.notifyError() {
  System.out.println("Some Observers not notified of change in Subject!");
}
```

```
  void Subject.updateError() {
    System.out.println("Observer not properly updated!");
  }

  void Subject.subjectMethodError() {
    System.out.println("Observers not notified of change in Subject!");
  }
```

5.3 JavaMOP Monitor

Besides the necessary *events*, the JavaMOP monitor in our Observer framework contains the specification of a *finite state machine*.

5.3.1 Events

All the JavaMOP events connect the pointcuts and advice actions which have been specified in the abstract aspect. Furthermore, a few events perform error handling.

```
ObserverMonitor(Subject subject) {
  event subjectEnrollment after(Subject subject) :
    ObserverAspect.subjectEnrollment(subject) {
      subject.afterSubjectEnrollment();
    }

  event beforeAttachObserver before(Subject subject, Observer observer) :
    ObserverAspect.attachObserver(subject,observer) {
      subject.beforeAttachObserver(observer);
    }

  event afterAttachObserver after(Subject subject, Observer observer) :
    ObserverAspect.attachObserver(subject,observer) {
      subject.afterAttachObserver(observer);
    }

  event beforeNotify before(Subject subject) :
    ObserverAspect.notify(subject) {
      subject.beforeNotify();
    }

  event notifyError before(Subject subject) :
    target(subject) && call(void Subject.notifyError()) {
      System.out.println("Error during notify");
    }

  event afterNotify after(Subject subject) :
```

```
  ObserverAspect.notify(subject) {
    subject.afterNotify();
  }

event beforeUpdate before(Subject subject, Observer observer) :
  ObserverAspect.update(subject, observer) {
    subject.beforeUpdate(observer);
  }

event updateError before(Subject subject) :
  target(subject) && call(void Subject.updateError()) {
    System.out.println("Error during update");
  }

event afterUpdate after(Subject subject, Observer observer) :
  ObserverAspect.update(subject, observer) {
    subject.afterUpdate(observer);
  }

event subjectMethod after(Subject subject) :
  ObserverAspect.subjectMethod(subject) {
    subject.afterSubjectMethod();
  }

event subjectMethodError before(Subject subject) :
  target(subject) && call(void Subject.subjectMethodError()) {
    System.out.println("Error during executing a subject method");
}
```

5.3.2 Finite State Machine

Our finite state machine contains only seven states. Let us introduce them, one after the other.

- `start`: In the initial state, the subject must be enrolled first, and then the state changes to `safe`. Before that, any other events are prohibited (unsafe).

  ```
  fsm :
    start [
      default start
      subjectEnrollment -> safe
      beforeNotify -> unsafe
      notifyError -> unsafe
      afterNotify -> unsafe
      beforeUpdate -> unsafe
  ```

```
  updateError -> unsafe
  afterUpdate -> unsafe
  subjectMethod -> unsafe
  subjectMethodError -> unsafe
]
```

- `safe`: In the state `safe`, it is permitted either to attach an observer, or to notify the observers, or to invoke any method of the subject.

```
safe [
  subjectEnrollment -> unsafe
  beforeAttachObserver -> attachingObserver
  afterAttachObserver -> unsafe
  beforeNotify -> notify
  notifyError -> unsafe
  afterNotify -> unsafe
  beforeUpdate -> unsafe
  updateError -> unsafe
  afterUpdate -> unsafe
  subjectMethod -> safe
  subjectMethodError -> unsafe
]
```

- `attachingObserver`: This state indicates that an observer is currently being attached to the subject. Two events are permitted to occur in this state:
  - `afterAttachObserver`, i.e., the attaching of the observer can be finished;
  - `beforeUpdate`, i.e., the observer can be updated before finishing its attaching.

```
attachingObserver [
  subjectEnrollment -> unsafe
  beforeAttachObserver -> unsafe
  afterAttachObserver -> safe
  beforeNotify -> unsafe
  notifyError -> unsafe
  afterNotify -> unsafe
  beforeUpdate -> updateDuringAttachingObserver
  updateError -> unsafe
  afterUpdate -> unsafe
  subjectMethod -> unsafe
  subjectMethodError -> unsafe
]
```

- `notify`: The subject is notifying all its observers. In this state, the `afterNotify` and the `updateDuringNotify` events are permitted.

```
notify [
  subjectEnrollment -> unsafe
  beforeAttachObserver -> unsafe
  afterAttachObserver -> unsafe
  beforeNotify -> unsafe
  notifyError -> unsafe
  afterNotify -> safe
  beforeUpdate -> updateDuringNotify
  updateError -> unsafe
  afterUpdate -> unsafe
  subjectMethod -> unsafe
  subjectMethodError -> unsafe
]
```

- `updateDuringAttachingObserver` and `updateDuringNotify`: An observer is updated during it is being attached to a subject or being notified. In these states, only the `afterUpdate` event is permitted.

```
updateDuringAttachingObserver [
  subjectEnrollment -> unsafe
  beforeAttachObserver -> unsafe
  afterAttachObserver -> unsafe
  beforeNotify -> unsafe
  notifyError -> unsafe
  afterNotify -> unsafe
  beforeUpdate -> unsafe
  updateError -> unsafe
  afterUpdate -> attachingObserver
  subjectMethod -> unsafe
  subjectMethodError -> unsafe
]
updateDuringNotify [
  subjectEnrollment -> unsafe
  beforeAttachObserver -> unsafe
  afterAttachObserver -> unsafe
  beforeNotify -> unsafe
  notifyError -> unsafe
  afterNotify -> unsafe
  beforeUpdate -> unsafe
  updateError -> unsafe
  afterUpdate -> notify
  subjectMethod -> unsafe
  subjectMethodError -> unsafe
]
```

- `unsafe`: This state announces that some pattern rule has been violated.

```
unsafe [
]
```

One may write some pieces of code to execute when a state occurs, e.g., to display messages, as follows.

```
@start {
  System.out.println("START");
}

@safe {
  System.out.println("SAFE");
}

@attachingObserver {
  System.out.println("ATTACHING OBSERVER");
}

@notify {
  System.out.println("NOTIFY");
}

@updateDuringAttachingObserver {
  System.out.println("UPDATE DURING ATTACHING OBSERVER");
}

@updateDuringNotify {
  System.out.println("UPDATE DURING NOTIFY");
}

@fail {
  System.out.println("FAIL");
  Runtime.getRuntime().exit(1);
}
```

6 Conclusion

In the previous sections, we made some investigations on AspectJ and JavaMOP in order to implement generic design pattern frameworks. We had to complete a lot of installation steps and to solve several installation/configuration problems, so you can also find our observations and advices in our report. Furthermore, we showed how to incorporate pattern-specific constraints as aspects and monitors in any Java program, through a comprehensive example.

A Case Study for the Observer Framework

Let us introduce a simple system that uses our Observer framework.

A.1 Classes

The instances of the `Time` class play the `Subject` role. A `Time` object records the time passing, by letting its `tickTock` method be called.

```
public class Time {
  protected int hour, minute, second;

  public Time() {
  }

  public Time(int hour, int minute, int second) {
    this.hour = hour;
    this.minute = minute;
    this.second = second;
  }

  public int getHour() {
    // Return hour in 12-hour mode.
    int h = hour % 12;
    return h > 0 ? h : 12;
  }

  public int getMinute() {
    return minute;
  }

  public int getSecond() {
    return second;
  }

  public boolean isAm() {
    return hour < 12;
  }

  public void tickTock() {
    second++;
    if (second == 60) {
      minute++;
      second = 0;
    }
    if (minute == 60) {
```

```
      hour++;
      minute = 0;
    }
    if (hour == 24)
      hour = 0;
  }
}
```

The instances of the `Clock` and `LazyPerson` classes play the *Observer* role, since they are to observe how time passing.

```
class Clock {
  protected int hour, minute;
  protected boolean isAm;

  public Clock(int hour, int minute, boolean isAm) {
    this.hour = hour;
    this.minute = minute;
    this.isAm = isAm;
  }

  public int getHour() {
    return hour;
  }

  public int getMinute() {
    return minute;
  }

  public boolean isAm() {
    return isAm;
  }
}

class LazyPerson {
  protected boolean isSleepy = true;

  public boolean isSleepy() {
    return isSleepy;
  }
}
```

A.2 Aspect

As it can be seen, the aforementioned classes contain no pattern-specific code. As it was proposed in Section 4, those codes are to be moved into the aspects.

A.2.1 Classes' Parents

First, each class has to have a role (`Subject` and `Observer`) assigned to. In AspectJ, this can be done in the following way:

```
public aspect TCLAspect extends ObserverAspect {

  declare parents: Time implements Subject;
  declare parents: Clock implements Observer;
  declare parents: LazyPerson implements Observer;
```

A.2.2 Implementing Abstract Auxiliary Methods

First, let us implement the state-specific methods. As it was mentioned in Section 5.2.2, the state of a subject or an object is represented by a string.

```
  public String Time.getState() {
    return "Time:" + hour + ":" + minute + ":" + second;
  }

  public String Clock.getState() {
    return "Clock:" + hour + ":" + minute + (isAm ? "am" : "pm");
  }

  public String LazyPerson.getState() {
    return "LazyPerson:" + (isSleepy ? "sleepy" : "ready");
  }
```

Implementing the `Subject.StateModified` and `Subject.StateConsistentWith` abstract methods is particularly an important issue since it is about how to decide when the subject should notify its observers.

```
  public boolean Time.StateModified(String previousState) {
    return !getState().equals(previousState);
  }

  public boolean Time.StateConsistentWith(String observerState) {
    if (observerState.startsWith("Clock:")) {
      String time = getState().substring("Time:".length());
      String clockTime = observerState.substring("Clock:".length());
      clockTime = clockTime.substring(0, clockTime.length() - 2);
      return time.startsWith(clockTime);
    } else if (observerState.startsWith("LazyPerson:")) {
      String time = getState().substring("Time:".length());
      int hour = Integer.parseInt(time.substring(0, time.indexOf(":")));
      String sleepyOrReady = observerState.substring("LazyPerson:".length());
      return (hour < 12 && sleepyOrReady.equals("sleepy"))
```

```
        || (hour >= 12 && sleepyOrReady.equals("ready"));
  }

  return false;
}
```

Next, the `Observer.update` abstract method must be implemented. Since both the `Clock` and `LazyPerson` classes have the `Observer` interface as parent, we propose two implementations:

```
public void Clock.update(Subject s) {
  Time t = (Time) s;
  hour = t.getHour();
  minute = t.getMinute();
  isAm = t.isAm();
}

public void LazyPerson.update(Subject s) {
  Time t = (Time) s;
  isSleepy = t.isAm();
}
```

A.2.3 Advice

Since any pattern-specific code is prohibited in the class `Time`, none of its methods may call a method which is pattern-specific. I.e., the only way to enforce such a behavior is to employ AspectJ.

The following advice calls the `Subject.notifyObs` methods whenever the `Time.tickTock` method has been called.

```
after(Time t): call(void Time.tickTock()) && target(t) {
  t.notifyObs();
}
```

A.3 Main Method

Let us implement a simple main method. As you can see, time is set to 11:59:58, and then the `Time.tickTock` method is called three times.

```
public static void main(String[] args) {
  Time aTime = new Time(11, 59, 58);

  Clock aClock = new Clock(12, 0, false);
  LazyPerson bob = new LazyPerson();

  aTime.attach(bob);
```

```
    aTime.attach(aClock);

    aTime.tickTock();
    aTime.tickTock();
    aTime.tickTock();

    if (bob.isSleepy()) {
      System.out.println("Too early for Bob!");
    } else {
      System.out.println("Bob is ready to face anotherday!");
    }
  }
```

When executing our program, one can notice, as it is expected, that Bob the lazy person is not sleepy any more. Furthermore, the framework has not fallen into an unsafe state. The output of the program:

```
SAFE
ATTACHING OBSERVER
before Update(Time:11:59:58,LazyPerson:sleepy)
UPDATE DURING ATTACHING OBSERVER
after Update(Time:11:59:58,LazyPerson:sleepy)
ATTACHING OBSERVER
SAFE
ATTACHING OBSERVER
before Update(Time:11:59:58,Clock:12:0pm)
UPDATE DURING ATTACHING OBSERVER
after Update(Time:11:59:58,Clock:11:59am)
ATTACHING OBSERVER
SAFE
before Notify(Time:11:59:59)
NOTIFY
before Update(Time:11:59:59,Clock:11:59am)
UPDATE DURING NOTIFY
after Update(Time:11:59:59,Clock:11:59am)
NOTIFY
before Update(Time:11:59:59,LazyPerson:sleepy)
UPDATE DURING NOTIFY
after Update(Time:11:59:59,LazyPerson:sleepy)
NOTIFY
after Notify(Time:11:59:59)
SAFE
after subjectMethod(Time:11:59:59)
SAFE
before Notify(Time:12:0:0)
NOTIFY
before Update(Time:12:0:0,Clock:11:59am)
```

```
UPDATE DURING NOTIFY
after Update(Time:12:0:0,Clock:12:0pm)
NOTIFY
before Update(Time:12:0:0,LazyPerson:sleepy)
UPDATE DURING NOTIFY
after Update(Time:12:0:0,LazyPerson:ready)
NOTIFY
after Notify(Time:12:0:0)
SAFE
after subjectMethod(Time:12:0:0)
SAFE
before Notify(Time:12:0:1)
NOTIFY
before Update(Time:12:0:1,Clock:12:0pm)
UPDATE DURING NOTIFY
after Update(Time:12:0:1,Clock:12:0pm)
NOTIFY
before Update(Time:12:0:1,LazyPerson:ready)
UPDATE DURING NOTIFY
after Update(Time:12:0:1,LazyPerson:ready)
NOTIFY
after Notify(Time:12:0:1)
SAFE
after subjectMethod(Time:12:0:1)
SAFE
Bob is ready to face anotherday!
```

References

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

[2] R. Miles, *AspectJ Cookbook*. O'Reilly, 2004.

[3] J. O. Hallstrom, N. Soundarajan, B. Tyler, *Monitoring Design Pattern Contracts*. Proceedings of the the 3rd FSE Workshop on the Specification and Verification of Component-Based Systems, pp. 87 - 93, 2004.

[4] AspectJ, http://www.eclipse.org/aspectj/

[5] AspectJ Development Tools (AJDT), http://www.eclipse.org/ajdt/

[6] F. Chen, G. Rosu, *MOP: An Efficient and Generic Runtime Verification Framework*. OOPSLA'07, pp. 569-588, 2007.

[7] Monitoring-Oriented Programming, JavaMOP,
http://fsl.cs.uiuc.edu/index.php/MOP/