Monitoring Design Pattern Contracts

Jason O. Hallstrom
Computer Science
Clemson University
Clemson, SC 29634, USA
jasonoh@cs.clemson.edu

Neelam Soundarajan, Benjamin Tyler
Computer Science and Engineering
Ohio State University
Columbus, OH 43210, USA
{neelam, tyler}@cse.ohio-state.edu

ABSTRACT

Design patterns allow system designers to reuse well established solutions to commonly occurring problems. These solutions are usually described informally. While such descriptions are certainly useful, to ensure that designers precisely and unambiguously understand the requirements that must be met when applying a given pattern, we also need formal characterizations of these requirements. Further, system designers need tools for determining whether a system implemented using a given pattern satisfies the appropriate requirements. In [18], we described an approach to specifying design patterns using formal *contracts*. In this paper, we develop a monitoring approach for determining whether the pattern contracts used in developing a system are respected at runtime.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Verification—*Runtime Monitoring*; D.1.m [**Programming Techniques**]: Patterns—*AOP*

General Terms

Design, Reliability, Verification

Keywords

Design patterns, Aspect-oriented programming, Runtime monitoring of contracts

1. INTRODUCTION

Design patterns [2, 8, 10, 17] have, over the last decade, fundamentally changed the way we think about the design of large software systems. Using design patterns not only helps designers exploit the community's collective wisdom and experience as captured in the patterns, it also enables others studying the system in question to gain a deeper understanding of how the system is structured, and why it behaves in particular ways. And as the system evolves over time, the patterns used in its construction provide guidance on managing the evolution so that the system remains faithful to its original design, ensuring that the original parts and the modified parts interact as expected. Although they are not components in the standard sense of the word, patterns may, as has been noted, be the real key to reuse since they allow the reuse of design, rather than mere code. But to fully realize these benefits, we must ensure that the designers have a thorough understanding of the precise requirements their system must meet in applying a given pattern, as well as automated or semi-automated ways of checking whether the requirements have been satisfied. To that end, the work we present in [18] describes an approach to specifying design patterns precisely using formal *contracts*. Our goal in this paper is to extend that work, and to develop a runtime monitoring approach that allows system designers to determine whether the patterns used in constructing a system have been applied correctly. We use an *aspect-oriented programming* [12, 11] approach to achieve this goal.

Consider the Observer pattern [8], illustrated in Fig. 1, which will be our case-study. There are two *roles* [15] in this pattern, Subject and Observer. The purpose of the pattern is to allow a set of objects that have enrolled to play the Observer role to be *notified* whenever the state of the object playing the Subject role changes, so that each of the observers[1] can update its state to be *consistent* with the new state of the subject. Also clear from Fig. 1 is the fact that the Notify() method of Subject will invoke the Update() method on each observer. What is not clear is when Notify() will be called and by whom. The informal description [8] states, "... subject notifies its observers whenever a change occurs that could make its observers' state inconsistent with

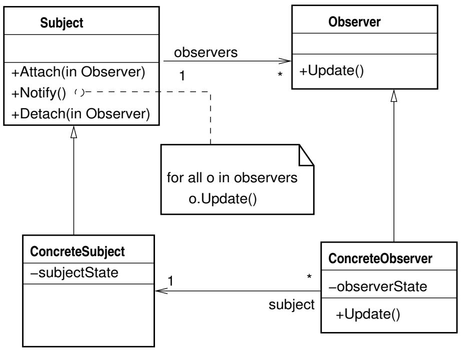

Figure 1: Observer Pattern

---

[1] We use names starting with uppercase letters, such as Subject, for roles; and lowercase names, such as subject, for the individual objects that play these roles. We also use names starting with uppercase letters for patterns. Occasionally, the name of a pattern is also used for one of its roles, as in the case of the Observer role of the Observer pattern. In such cases, the context will make clear whether we are talking about the role or the pattern.

its own." But it is not clear how the subject will know when its state has become inconsistent with that of one or more observers. Indeed, what does it mean to say that the subject state has become *inconsistent* with that of an observer? In other words, what exactly are the requirements that the designer must ensure are met in order to apply this pattern as intended? The pattern contracts described in [18] provide precise answers to these questions. We will consider the requirements specified by these contracts in Section 2.

Next consider the question of runtime monitoring. In standard specification-based testing/monitoring [1, 3, 13], we typically consider the behavior of the methods of a *single* class. For this, we instrument the class in question to see that the pre- and post-conditions of the class methods are satisfied at appropriate points. But in the case of patterns, we are dealing not with individual classes, but with multiple classes. Indeed, the focus is usually on the interactions and interrelations among the classes, rather than on the behaviors of the classes in isolation.

A natural solution is to use *aspects* [12, 11], since aspects allow us to deal with *crosscutting* concerns. We will define an *abstract aspect* for Observer that implements the monitoring functionality common across all applications of the pattern. Corresponding to any particular application in an actual system, we will define a *concrete subaspect* that tailors the monitoring functionality as appropriate to the application in question. An abstract aspect captures the requirements embodied in a given pattern's *contract*, and a concrete subaspect captures the specializations embodied in a *subcontract* of the pattern's contract. In a sense, we can consider the aspects used in the current paper as aspect-versions of the contracts presented in [18]. (Although, [18] did not consider the notion of a subcontract.) Indeed, hereafter, we refer to the abstract aspect as a *contract*, and the concrete subaspect as a *subcontract*. We will see how a contract-subcontract pair can be used to monitor a system to see if it applies a given pattern faithfully.

There is an inherent risk in formalizing patterns in that their hallmark flexibility may be lost [16]. For the case of Observer, if we adopt one definition for the notion of *consistency* between the Subject and Observer states, the pattern may not be usable in systems that have a different notion of this concept; or we may have to come up with multiple contracts, one for each possible notion of consistency. Clearly, this would be undesirable. As we will see, our contract for Observer, while precisely capturing the pattern requirements, will also retain all of the flexibility contained in the pattern.

Hannemann and Kiczales [9] show how patterns can be *implemented* as aspects. They argue that the code for a given pattern should be collected within an aspect, rather than being distributed among different classes. By contrast, the aspect we develop in Section 2 *monitors* a system to check whether it satisfies the requirements that any designer implementing the Observer pattern must meet. This raises the question, does the Hannemann-Kiczales implementation of the pattern meet our contract? It must, if our contract is truly general. As we will show, we can indeed define a subcontract for this implementation of Observer, in exactly the same way as we do for a more 'standard' implementation of the pattern in Section 3. This is remarkable because when developing the contract for Observer, we only had in mind standard class-based implementations of the pattern, and we tried to ensure that our contract would be appropriate for all such implementations. Here we had a very different kind of implementation, and our contract turned out to be appropriate for this implementation as well. Further, and somewhat to our surprise, when we ran our contract and subcontract against this aspect-based implementation of the pattern, a contract violation was reported! It turns out, as we will see, that there is a minor error in the implementation in [9].

In Section 2, we develop the contract for Observer, present a simple system built using the pattern, define the subaspect corresponding to the pattern as used in this system, show how the aspect and subaspect allow us to monitor the system at runtime, and discuss the monitoring results. In Section 3, we outline how a subaspect corresponding to the implementation of [9] can be defined, and discuss the monitoring results. In the next section, we discuss related work. In Section 5, we summarize our approach, and provide pointers to future work.

2. PATTERN CONTRACTS

Since we use *AspectJ* [11] to develop the pattern contracts and subcontracts, we begin with a brief summary of some of the essential parts of *AspectJ*. Three key concepts of the language are join points, pointcuts, and advice. A *join point* identifies a particular point in the execution of a program; for example, a call to a particular method of a particular class, or a call to a particular constructor of a particular class. A *pointcut* is a way of grouping together a set of join points that we want to treat in a particular fashion; for example, calls to *all* methods of a given class. The pointcut construct enables us to collect *context*: for example, the object on which the method in question was applied, or the additional arguments that were passed to the method. Finally, the *advice* associated with a given pointcut specifies the code that needs to be executed at runtime when control reaches any of the join points that match the pointcut. There are three distinct types of advice. Consider a method call. The associated *before* advice, if any, will be executed before the method is executed. The *after* advice, if any, will be executed after the method is executed. We do not use the third kind of advice, the *around* advice.

2.1 Observer Contract

The aspect that defines the Observer contract appears in Figures 2, 3, and 4. The following notes explain the lines with the corresponding numbers in the figures.

1. The interfaces Subject and Observer correspond to the two roles of the pattern. Note that unlike in Fig. 1, there are no methods such as Notify() in these interfaces. The pointcuts of *AspectJ*, as we will see, provide a more general way of introducing these.

2. ObserverPatternContract, an *abstract aspect*, captures the requirements to be checked for all applications of Observer.

3. The information needed to monitor the system will be maintained in three variables: xSubjectsObservers maps[2] each object that enrolls as a subject to the objects that are enrolled to be observers of that subject,

---

[2] This should be a WeakHashMap to allow garbage collection to proceed normally.

```
protected interface Subject { }                      // see note (1)
protected interface Observer { }
public abstract aspect ObserverPatternContract {          note (2)
  //Aux. variables:                                           (3)
   private Map xSubjectsObservers = new HashMap();
   private Set xUpdateCalls = new HashSet();
   private Map xrecordedStates = new HashMap();
  //Auxiliary functions:                                      (4)
   abstract protected String xSubjectState(Subject s);
   abstract protected String xObserverState(Observer o);
   abstract protected boolean
         xModified(String s1, String s2);
   abstract protected boolean
         xConsistent(String s, String o);
  //Pointcuts:                                                (5)
   abstract protected pointcut subjectEnrollment(Subject s);
   abstract protected pointcut
         attachObs(Subject s, Observer o);
   abstract protected pointcut
         detachObs(Subject s, Observer o);
   abstract protected pointcut Notify(Subject s);
   abstract protected pointcut Update(Subject s,Observer o);
   abstract protected pointcut subjectMethods(Subject s);
```

Figure 2: Observer Contract (part 1 of 3)

and is initially empty. xUpdateCalls is used to keep track of the observers that are updated when the corresponding subject state changes. xrecordedStates is used to save, for each subject, the state that its observers have been most recently notified of.

4. We use auxiliary functions to represent pattern concepts that vary among applications. As we noted earlier, the pattern requires that observers become *consistent* with the subject state when they are updated, but the notion of consistency will vary from one system to another. Similarly, the pattern requires the observers to be notified when the subject state is *modified*, but what modification of the subject state means will vary from system to system. xModified() and xConsistent() allow us to specify these requirements precisely, while allowing for variation among different systems.

   Given two subject states, xModified() tells us if the second state should be considered 'modified' from the first. Since this function is *abstract*, the pattern contract will not define it; instead, the subcontract will provide a definition tailored to the system in question. Similarly, xConsistent(), given a subject state and an observer state, tells us whether the latter is *consistent* with the former. This, too, is abstract, since the notion of consistency varies from system to system.

   For simplicity, rather than working with the actual states of the subjects and observers, we assume that we have functions xSubjectState() and xObserverState() that will encode the states into Strings. Naturally, such encodings will depend on the system: hence these are *abstract*, to be suitably defined in the subcontract.

5. Next we have the pointcuts that identify the points at which the system should be *interrupted* at runtime, either to save information needed by the contract, or to check if the contract requirements are being met.

   subjectEnrollment is the pointcut that represents the points at which an object enrolls to play the Subject role. The only argument here is the object enrolling. attachObs and detachObs correspond to an object attaching or detaching, respectively. The arguments for these two pointcuts are the subject and observer involved.

   Next we have Notify, which corresponds to the points at which a given subject's observers are notified following a change in the state of the subject (as defined by the xModified() function). The Update pointcut corresponds to an individual observer being updated to become consistent (as defined by xConsistent()) with the modified (or rather, xModified()) subject state. The final pointcut, subjectMethods, corresponds to all the methods of the class playing the Subject role.

```
  //Advice for Subject enrollment:                            (6)
   after(Subject s): subjectEnrollment(s) {
     Set obSet = new HashSet();
     xSubjectsObservers.put(s,obSet);
     xrecordedStates.put(s,xSubjectState(s)); }
  //Advice for attaching Observer:                            (7)
   before(Subject s, Observer o): attachObs(s,o) {
     xUpdateCalls.clear(); }
   after(Subject s, Observer o): attachObs(s,o) {
     if (!xUpdateCalls.contains(o)) { System.out.println(
     "Update not called on attaching Observer"); }
     Set obSet = (Set)xSubjectsObservers.get(s);
     obSet.add(o); xSubjectsObservers.put(s,obSet); }
  //Advice for detaching Observer:                            (8)
   before(Subject s, Observer o): detachObs(s,o) {
     Set obSet = (Set)xSubjectsObservers.get(s);
     obSet.remove(o); xSubjectsObservers.put(s,obSet); }
   //No "after" advice for detachObs.
```

Figure 3: Observer Contract (part 2 of 3)

Let us now consider the advice corresponding to the various pointcuts[3].

6. The advice for subjectEnrollment adds the enrolling object to xSubjectsObservers with an empty set of observers, and saves its current state as its recorded state. As there are no observers for this subject, we can vacuously say that they have all been informed of its current state.

7. When a new observer attaches to a subject, we must ensure that it is updated. As we noted in [18], this point has been overlooked in many informal descriptions of the pattern. If this is not done, the observer's state may be inconsistent with the subject state until the point when the subject is next modified.

   To check this, the before advice clears xUpdateCalls. As we will see below, the advice for the Update pointcut adds the observer being updated to xUpdateCalls.

---

[3] Java's collection classes rely on Object.equals() to locate items. We assume the default implementation of equals(), which tests for equality based on the *identity* of the objects.

Hence, in the after advice of attachObs, we require xUpdateCalls to contain this observer. If it does not, that indicates that the observer was *not* updated when it enrolled, and we output a message to that effect[4].

8. Detachment of an observer simply requires eliminating it from the set of objects enrolled to observe the subject. It is possible that in the actual system, nothing is done at this point, i.e., the designer might have decided to continue updating the object whenever the subject's state is modified. This will not violate our contract; and it is consistent with the intent of the pattern since the pattern requires that all enrolled observers be updated, not that others should *not* be[5].

```
//Advice for Notify:                                        (9)
  before (Subject s) : Notify(s) {
    xUpdateCalls.clear();
    xrecordedStates.put(s,xSubjectState(s)); }
  after (Subject s) : Notify(s) {
    Set obSet = (Set)xSubjectsObservers.get(s);
    if (!xUpdateCalls.containsAll(obSet)) {
      System.out.println("Some Observers not notified
          of change in Subject!"); }   }
//Advice for Update:                                        (10)
  before (Subject s, Observer o) : Update(s,o)
    { xUpdateCalls.add(o); }
  after (Subject s, Observer o) : Update(s,o) {
    if (!xConsistent(xrecordedStates.get(s),
          xObserverState(o))) { System.out.println(
          "Observer not properly updated!"); }   }
//Advice for Subject's methods:                             (11)
  after(Subject s): subjectMethods(s) {
    if (xModified(xrecordedStates.get(s),xSubjectState(s))){
      System.out.println("Observers not notified
          of change in Subject!"); }    }
}
```

Figure 4: Observer Contract (part 3 of 3)

9. The before advice for Notify updates xrecordedStates for the subject since its observers are about to be notified of its state change. And xUpdateCalls is cleared so in the after advice we can check that *all* of its observers have been notified. If not, we print a suitable message.

10. The before advice of Update adds the observer to the set of observers being updated. In the after advice, we check that the state of the observer is consistent with the subject state. This checks that the system code that is supposed to update the observer is working correctly, at least as judged by the definition of xConsistent(). If the condition is not satisfied, it may be an error in the subcontract, rather in the monitored system. We clearly need to identify such errors and correct them, and such checks help with that task.

11. The final advice corresponds to the methods of the class playing the role of Subject. For any such method, there are three possibilities.

    First, the method execution did not change the subject state (according to xModified()). Hence, the final state should match the recorded state of the subject, assuming that this condition was satisfied at the start of the method. (If this were not the case, an earlier error would already have been caught.)

    Second, the method execution changed the subject state and called the appropriate operations to notify/update the observers. This would have triggered the advice associated with the Notify pointcut, and the advice associated with Update for each observer. Those two advices would have checked that all observers were updated, and would also have saved, in xrecordedStates, the state of the subject at that time. So the final subject state would match that in xrecordedStates.

    Third, the method changed the subject state, but did not notify the observers. Or perhaps the method changed the subject state, notified the observers, and then *again* changed the subject state, and this time did not notify the observers. In both cases, the if-condition of the after advice would be true, and we would get the appropriate error message.

It is worth stressing that by specifying the auxiliary functions and pointcuts as *abstract*, we have ensured that all of these can be defined, in the subcontract, as appropriate to the particular system. But at the same time, the checks in the various pieces of advice ensure that the essential intent of the pattern is not violated. Thus, the contract precisely specifies the pattern's requirements without in any way compromising flexibility.

2.2 A Simple System Using Observer

Fig. 5 presents TCL, a simple system that uses Observer. Instances of the Time class play the Subject role. Instances of Clock and LazyPerson play the Observer role; these two classes implement the TimeObserver interface. The Time class maintains a hash set of objects that enroll (via its attach() method) to observe the time. When the time changes, which only happens in the tickTock() method, the object calls its notifyObs() operation, which invokes the update() operation on each of its observers. In the main() method, we create aTime (a Time object), aClock (a Clock object), and bob (a LazyPerson object), attach the latter two to aTime, invoke tickTock() a few times on aTime, and then check the state of bob. TCL is a fairly standard, if simple, example of a system built using the Observer pattern.

2.3 Observer Subcontract for TCL

The subaspect, appropriate to TCL, that defines the subcontract of our pattern contract appears in Fig. 6[6].

12. We use the declare parents mechanism of *AspectJ* to state that Time implements the Subject interface of

---

[4] We should note that in our actual contract, we have additional checks. For example, in the before advice for this pointcut, we check that this object has not already enrolled as an observer for this subject. We also check that s has enrolled as a Subject. We omit some of these details.

[5] If we wish to disallow detached observers from being updated, the contract can be suitably modified: in the after advice of Notify, check that obSet.containsAll(xUpdateCalls) evaluates to true; i.e., for any subject, the set of updated observers must equal the set of attached observers.

[6] For readability, we use "∧", rather than the standard "&&", to denote the 'and' operation.

```
interface TimeObserver { public void update(Time t); }
class Clock implements TimeObserver {
  protected int hour = 12, minute = 0;
  public void update(Time t) {
    hour = t.getHour(); minute = t.getMinute(); }
  public String ClockTime() {
    return("The time is: " + hour + ":" + minute); }
}
class LazyPerson implements TimeObserver {
    protected boolean isSleepy = true;
    public void update(Time t) { isSleepy = t.isAm(); }
    public boolean readyToRiseNShine(){ return (!isSleepy); }
}
class Time {
  protected HashSet observers = new HashSet();
  protected int hour = 0, minute = 0, second = 0;
  public void attach(TimeObserver o) {
     observers.add(o); o.update(this); }
  public void detach(TimeObserver o) { observers.remove(o);}
  protected void notifyObs() {
    for (Iterator e = observers.iterator() ; e.hasNext() ;) {
       ((TimeObserver)e.next()).update(this); } }
  public int getHour() { // Return hour in 12-hour mode. }
  public int getMinute() { . . . }
  public int getSecond() { . . . }
  public boolean isAm() { . . . }
  public void tickTock() {
    // Update hour, etc. appropriately. Code omitted.
    // In our actual system, this function sets the Time to
    // a random (legal) value.
    notifyObs(); }
  public static void main(String[] args) {
    Time aTime = new Time(); Clock aClock = new Clock();
    LazyPerson bob = new LazyPerson();
    aTime.attach(bob); aTime.attach(aclock);
    aTime.tickTock(); aTime.tickTock(); aTime.tickTock();
    System.out.println(aClock.ClockTime());
    if (bob.readyToRiseNShine()) {
      System.out.println("Bob is ready to face another day!");}
    else { System.out.println("Too early for Bob!"); }  }
}
```

Figure 5: Time-Clock-LazyPerson (TCL) System

the pattern contract (Fig. 2), and that TimeObserver is an extension of the Subject interface.

13. Next we provide definitions for the abstract pointcuts of the base contract. Thus, attachObs is defined as a call to the attach() method of Time, since that is the method that Time's observers are required to use to enroll as observers. detachObs, Notify, and Update are equally direct. In each case, we use the target and args constructs of *AspectJ* to bind the parameters of the pointcut with the appropriate entities from the actual (join) point in the system.

    In TCL, there is no explicit enrollment of a Time object as a subject; instead, it becomes a subject upon construction. We define the subjectEnrollment pointcut accordingly. subjectMethods captures *all* the methods of the Time class. Note that if in a future modification of the system, new methods are added to Time, those methods will also be captured by this pointcut, and will be required to abide by the requirements of the pattern contract, as captured by clause (11) in Fig. 4.

```
public aspect TCLContract extends ObserverPatternContract{
  declare parents: Time implements Subject;                (12)
  declare parents: TimeObserver extends Observer;
  //Pointcuts:                                             (13)
    protected pointcut attachObs(Subject s, Observer o):
      call(void Time.attach(TimeObserver))
            ∧ target(s) ∧ args(o);
    protected pointcut detachObs(Subject s, Observer o):
      call(void Time.detach(TimeObserver))
            ∧ target(s) ∧ args(o);
    protected pointcut subjectEnrollment(Subject s):
      call(Time.new()) ∧ target(s);
    protected pointcut subjectMethods(Subject s):
      call(* Time.*()) ∧ target(s);
    protected pointcut Notify(Subject s):
      call(void Time.notifyObs()) ∧ target(s);
    protected pointcut Update(Subject s, Observer o):
      call(void TimeObserver.update(Time))
            ∧ target(o) ∧ args(s);
  //Aux. functions:                                        (14)
  protected String xSubjectState(Subject s) {
    //s must be of type Time; return the time as a String. }
  protected String xObserverState(Observer o) {
    //o must be of type Clock or LazyPerson; use getClass()
    //to check, and return state encoded as a String. }
  protected boolean xModified(String s1, String s2) {
    //Return true if the times encoded in s1 and s2 are
    // equal, else false. }
  protected boolean xConsistent(String s, String o) {
    //Check if o encodes a Clock state or a LazyPerson state.
    //For a LazyPerson, return true if isSleepy agrees with hour
    //in the Time state encoded in s being between 0 and 11.
    // Similarly if o encodes a Clock. }
}
```

Figure 6: TCL Subcontract

14. Next we define the auxiliary functions. xSubjectState() encodes the time represented by the the given Time object. xObserverState() is similar, but has to handle two types of observer objects, Clock and LazyPerson. xModified() determines whether the times encoded in its two arguments are equal. xConsistent(), depending on whether the state encoded in the second argument is of type Clock or LazyPerson, compares the value of either isSleepy, or hour and minute in that argument to the time in the first argument.

These definitions are dictated by the TCL system. If we considered another system that had different classes playing the Subject and/or Observer roles, or did the *notification*, *update*, etc. in other ways, we would have to define another subcontract tailored to that system. But for another system that uses the same classes as TCL, and does the notification, etc., in the same manner as TCL, we can use the same subcontract.

2.4 Results of Runtime Monitoring

We can now compile the abstract aspect that captures the Observer contract (Figs. 2, 3, 4), the subaspect that captures the subcontract for this system (Fig. 6), and the actual system code (Fig. 5) using the *AspectJ* compiler. The compiler will do the necessary *code weaving* [12, 11]. When the resulting byte code is executed, if there are no problems, that is, if all the requirements of the pattern contract/subcontract are met, the system will run as usual (if a bit slower than usual). However, in order to check that the monitoring was indeed progressing appropriately, we inserted additional output statements in the various pieces of advice, as well as in the tickTock() method, to help us track the progress of the system. A portion of the output from a sample run appears in Fig. 7 (the line numbers were inserted by hand).

```
1:  Tick-tock!
2:     before Notify(Time:11:42:06)
3:     before Update(Time:11:42:06, Clock:5:48am)
4:     after subjectMethods(Time:11:42:06)
5:     after subjectMethods(Time:11:42:06)
6:     after subjectMethods(Time:11:42:06)
7:     after Update(Time:11:42:06, Clock:11:42am)
8:     before Update(Time:11:42:06, LazyPerson:true)
9:     after Update(Time:11:42:06, LazyPerson:true)
10:    after Notify(Time:11:42:06)
11:    after subjectMethods(Time:11:42:06)
12: Tick-tock!
13:    before Notify(Time:17:09:06)
14:    . . .
19:    before Update(Time:17:09:06, LazyPerson:true)
20:    after Update(Time:17:09:06, LazyPerson:true)
21:    *** Observer not properly updated!
22:    * Subject: Time:17:09:06; Observer: LazyPerson:true
```

Figure 7: Sample Monitored Run of TCL System

Line 1 indicates that tickTock() was called, which resulted in Time.notifyObs() being called, which resulted in the Notify pointcut being entered, with the aTime value at this point being as stated (line 2). Next (line 3), Update on aClock was called. (Note that the clock reading is incorrect in this line because we have not yet done the update.) Updating aClock requires three calls to the Time methods for getting the hour, minute, and am/pm information. In each case, the after advice of the subjectMethods pointcut was executed. The advice did not report any problems, since at the start of Notify, xrecordedStates had already been updated for this Time object. The outputs from the after advice for these three calls appear in lines 4, 5, and 6. Finally, the update() operation finished, and the output from the after advice (line 7) shows that the clock was properly updated.

Next, notifyObs invoked update() on the bob object. During this run, we inserted an error in the system by replacing the code of LazyPerson.update() with an empty body; this update() operation did not invoke any operation of Time. Hence, immediately following the output from the before advice of Update (line 8), we have the output from the after advice (line 9). But there was no error reported, because the value of bob.isSleepy happened to have the correct value. In the next call to tickTock(), the error was reported (lines 21, 22). Thus, without any changes in the code of TCL, we were able to monitor the system to see if it met the appropriate pattern requirements. For a more complex system built using several patterns, we would define the appropriate contract and subcontract for each, and would compile all of them against the system source code.

3. MONITORING ALTERNATE PATTERN IMPLEMENTATIONS

As required by the pattern, notifyObs() in Time, and update() in Clock and LazyPerson, are all concerned with updating the observers when the state of the Time object changes. Hannemann and Kiczales [9] argue that such code is better written as an aspect, thereby *localizing* this code in a single module. They present an aspect that implements Observer. The aspect contains the code for *notifying* the observers of a given subject when the subject state changes. This naturally involves calling an update() operation on each observer; this operation is flagged as abstract since it will depend on the class of the observer. Further, they define an abstract pointcut, subjectChange, intended to capture all the methods of the Subject class that might result in the subject state being modified. This portion of their aspect looks as in Fig. 8.

```
abstract protected pointcut subjectChange(Subject s);
abstract protected void updateObserver(
  Subject s, Observer o);
after (Subject s): subjectChange(s) { notifyHandler(s); }
public void notifyHandler(Subject s) {
  Iterator i = ((Set)perSubjectObservers.get(s)).iterator();
  if (i==null) { System.out.println("Trouble 1"); }
  else { while (i.hasNext()) {
          updateObserver(s, (Observer)i.next()); } }
```

Figure 8: Partial AOP Implementation of Observer

We have made a slight change in their code; we have written the after advice for subjectChange as a call to notifyHandler(). In the original version, notifyHandler() is not introduced; instead, the advice simply contains the code that appears in the body of our notifyHandler(). The reason for this change is that in defining the subcontract corresponding to this implementation of Observer, we need to define the execution of this after advice as our Notify pointcut, but *AspectJ* does not provide a construct that will allow us to do so[7]. Therefore, we introduce the notifyHandler() method corresponding to this advice, and use this method to define the Notify pointcut.

The aspect in [9] also defines the code shown in Fig. 9, for adding and removing an observer. The code for adding an observer adds the object to the set corresponding to the subject; the code for removing an observer removes it from this set. Here, too, we have made a change. If the map does not contain an entry for the subject, that means the object is not currently enrolled. We must then add it (paired with a set consisting of just this observer) to the map. This is the point where the object is enrolling as a Subject. So this point should, in our subcontract, be captured by the subjectEnrollment pointcut. To achieve this, we have introduced an empty method, subEnroll(), inserted a call to it in

---

[7] Recent versions of *AspectJ* seem to include such constructs.

```
public void addObserver(Subject s, Observer o) {
  Set obSet = (Set)perSubjectObservers.get(s);
  if (obSet == null) {obSet = new HashSet(); subEnroll(s);}
  obSet.add(o); perSubjectObservers.put(s,obSet); }
public void removeObserver(Subject s, Observer o) {
  Set obSet = (Set)perSubjectObservers.get(s);
  obSet.remove(o); perSubjectObservers.put(s,obSet); }
public void subEnroll(Subject s) { ; }
```

Figure 9: AOP Implementation of Observer (cont'd)

addObserver(), and will define the subjectEnrollment pointcut (in the subaspect) as a call to subEnroll().

Let us now turn to the subcontract, presented in Fig. 10, corresponding to this implementation of Observer. Due to space limitations, we present only some key portions of the subaspect.

```
protected pointcut attachObs(Subject s, Observer o):
  call(void HKObserver.addObserver(Subject, Observer))
        ∧ args(s,o);
protected pointcut subjectEnrollment(Subject s):
  call(void HKObserver.subEnroll(Subject)) ∧ args(s);
protected pointcut Notify(Subject s):
  call(void HKObserver.notifyHandler(Subject))
        ∧ args(s);
```

Figure 10: Subcontract for AOP Implementation

As we noted above, introducing the subEnroll() method allows us to define an appropriate pointcut for subject enrollment. Similarly, introducing notifyHandler() allows us to define the Notify pointcut. The attachObs pointcut is defined directly in terms of the addObserver() method.

We next ran this implementation (along with the concrete Subject and Observer classes defined in [9]) using our pattern contract and subcontract. Surprisingly, the system printed a message indicating that an observer was not properly updated. Further analysis showed that the addObserver() code (Fig. 9) does not meet the requirement of the pattern contract (Fig. 3, line (7)) that requires observers to be updated upon attachment. Thus, our original contract is general enough to be used to monitor such novel implementations of patterns.

4. RELATED WORK

A number of authors have recognized the importance of describing patterns precisely. The work in [20, 4], for example, improves the traceability of design patterns in design documentation by developing UML extensions. Other authors have more directly addressed the requirements question. Eden *et al.* use a higher-order logic formalism [7, 5] to encode patterns as formulae. The primitives of the logic include classes, methods, and the relations among them. While the approach seems to capture the structural properties of interest, it provides only limited support for behavioral properties. Mikkonen [14] specifies behavioral properties of patterns using an action system, the guarded commands of which operate over abstract models and relations. Taibi *et al.* combine these two approaches to capture both structural and behavioral properties.

There does not seem to be much work focused explicitly on monitoring design pattern specifications. In [19], the authors discuss issues in testing software created using patterns that rely heavily on the use of dynamic binding and dynamic dispatch, but the question of testing whether the patterns are being used correctly is not considered. Techniques for *implementing* design patterns may be worth mentioning. Much of this work targets the development of pattern repositories encoding individual patterns that can be applied to an existing design automatically [6, 21]. More relevant to our work, however, is the aspect-based implementation approach of Hanneman and Kiczales [9] discussed earlier.

5. DISCUSSION

The goal of our work was to develop a monitoring approach for determining whether design pattern requirements are satisfied at runtime. As patterns cut across class boundaries, the requirements to be checked are also cross-cutting. An AOP-based approach was therefore a natural choice. The monitoring code common across all applications of a given pattern is implemented as an abstract aspect; the parts that vary among applications are expressed over abstract functions and pointcuts. These functions and pointcuts are defined in a subaspect corresponding to a particular application of the pattern. The abstract aspect and subaspect combined form the complete monitoring code for the system in question.

Our monitors are fairly robust. Consider, for example, the requirements defined for subjectMethods. Suppose a designer, as part of evolving a system, adds a new method to the class that plays the Subject role, and that this method modifies the state of the object. Even if the new method respects the invariants of the class, problems will arise if the designer neglects to call notifyObs() after performing the modifications, as this will leave the object inconsistent with its observers. Such maintenance errors will be detected by monitoring the new system without any changes to our aspect-based monitor.

Our future work aims to investigate the applicability of our monitoring approach to other types of design patterns. In particular, we plan to investigate more complex patterns, such as those used in concurrent and networked systems.

6. ACKNOWLEDGEMENTS

We would like to thank the anonymous referees for their detailed comments on the first draft of this paper.

7. REFERENCES

[1] R. Binder. *Testing object-oriented systems.* Addison-Wesley, 1999.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: A system of patterns.* Wiley, 1996.

[3] Y. Cheon and G. Leavens. A simple and practical approach to unit testing: The jml and junit way. In *Proc. of ECOOP 2002*, pages 231–255. Springer-Verlag LNCS, 2002.

[4] J. Dong. UML extensions for pattern compositions. *J. of Object Technology*, 3:149–161, 2002.

[5] A. Eden. A visual formalism for object-oriented architecture. In *Proceedings, Integrated Design and Process Technology (IDPT-2002)*, June 2002.

[6] A. Eden, J. Gil, Y. Hirshfeld, and A. Yehudai. Toward a mathematical foundation for design patterns. Technical Report 004, Tel Aviv University, 1999.

[7] A. Eden, A. Yehudai, and J. Gil. Precise specification and automatic application of design patterns. In *Automated Software Engineering*, pages 143–152, 1997.

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

[9] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proc. of OOPSLA*, pages 161–173. ACM, 2002.

[10] R. Johnson. Components, frameworks, patterns. In *Symposium on Software Reusability*, 1997.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proc. 15th ECOOP*, pages 327–353. Springer, 2001.

[12] C. Lopes, B. Tekinerdogan, W. de Meuter, and G. Kiczales. Aspect oriented programming. In *Proc. of ECOOP'98*. Springer, 1998.

[13] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

[14] T. Mikkonen. Formalizing design patterns. In *Proceedings of 20th ICSE*, pages 115–124. IEEE Computer Society Press, 1998.

[15] T. Reenskaug. *Working with objects.* Prentice-Hall, 1996.

[16] D. Riehle. Composite design patterns. In *Proc. of OOPSLA*, pages 218–228. ACM, 1997.

[17] D. Riehle and H. Zullighoven. Understanding and using patterns in software development. *Theory and Practice of Object Systems*, 2(1):3–13, 1996.

[18] N. Soundarajan and J. Hallstrom. Responsibilities and rewards: specifying design patterns. In *Proc. of Int. Conf. on Software Engineering (ICSE)*, 2004.

[19] W. Tsai, Y. Tu, W. Shao, and E. Ebner. Testing extensible design patterns in object-oriented frameworks through scenario templates. In *Proc. of COMPSAC*, pages 166–171, 1999.

[20] J. Vlissides. Notation, notation, notation. *C++ Report*, April 1998.

[21] S. Yau and N. Dong. Integration in component-based software development using design patterns. In *24th Ann. Int. Computer Software Applications Conf.*, Taipei, Taiwan, October 2000.