Amplifying the Benefits of Design Patterns: From Specification Through Implementation

Jason O. Hallstrom[1], Neelam Soundarajan[2], and Benjamin Tyler[2]

[1] Department of Computer Science, Clemson University
jasonoh@cs.clemson.edu
[2] Department of Computer Science and Engineering, Ohio State University
{neelam, tyler}@cse.ohio-state.edu

Abstract. The benefits of design patterns are well-established. We argue that these benefits can be further amplified across the system lifecycle. We present two contributions. First, we describe an approach to complementing existing informal pattern descriptions with precise pattern specifications. Our specification language captures the properties common across all applications of a pattern, while accommodating the variation that occurs across those applications. Second, we describe an approach to monitoring a system's runtime behavior to determine whether the appropriate pattern specifications are respected. The monitoring code is generated automatically from the pattern specifications underlying the system's design. We conclude with a discussion of how our contributions are beneficial across the software lifecycle.

1 Introduction

Design patterns [1–3] have become an important part of software practice, fundamentally impacting the design of commercial systems, class libraries, etc. Patterns capture the distilled wisdom of design communities by describing a set of recurring problems, proven solutions to those problems, and the conditions under which the solutions can be applied. They are usually presented as part of a *catalog* that includes a set of patterns relevant to a particular problem domain or application area. When a designer is faced with a design difficulty, the relevant catalogs provide guidance on how to address the difficulty. This idea continues to gain influence; patterns are being discovered and applied in emerging areas as diverse as wireless sensor network design and bioinformatics.

But the benefits of patterns are undercut by three important factors. First, although the informal style used in current pattern catalogs has proven useful, it creates a potential for ambiguity and misunderstanding that jeopardizes the correct use of patterns. This is likely to be a serious problem for team-based projects since different interpretations of a pattern are likely to manifest themselves as incompatibilities among different parts of a system. Second, there is insufficient tool support to assist in discovering pattern implementation errors. Again, these types of tools are especially relevant to team-based projects, where they could be used to detect inconsistent pattern applications. Third, changes

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 214–229, 2006.

introduced during system evolution and maintenance may erode the pattern applications underlying the original design, compromising the *design integrity* of the modified system. The goal of our work is to address these issues, amplifying the benefits of design patterns across the software lifecycle.

We present two contributions. First, we present a *pattern contract language* that captures the structural and behavioral requirements associated with a range of patterns, as well as the system properties that are guaranteed as a result. In addition, the contract language supports *subcontracts*, a form of contract refinement that allows system designers to capture, in a precise way, the customizations made to particular patterns when they are applied. This language will be used to develop *contract catalogs* that complement existing informal pattern catalogs. Second, we present an approach to monitoring a system's runtime behavior to determine whether the system abides by the relevant pattern requirements. The monitoring code for a given system is generated automatically based on the pattern contracts and subcontracts underlying its design.

Before we proceed, it is important to consider a potential problem introduced by developing pattern descriptions that are *precise*. One might argue that existing descriptions are intentionally ambiguous to support flexibility in how patterns may be applied. Precision and flexibility might be at odds here. As we will see, this is not the case. Our approach makes it possible to achieve precision without compromising flexibility. Indeed, in our experience, the process of developing precise descriptions often leads to the discovery of new dimensions of flexibility that are not evident in the informal descriptions.

The rest of the paper is organized as follows. In Section 2, we present a simple pattern-based system, and discuss the difficulties that might be encountered by a software team developing this system. It serves as a running example throughout the paper. In Sections 3 and 4, we present our contract language and contract monitoring approach, respectively. In Section 5, we discuss elements of related work. Finally, in Section 6, we conclude with a summary of our contributions, their benefits to the system lifecycle, and provide pointers to future work.

2 A Pattern-Based Design

To motivate the problems that our work addresses, consider developing a basic simulation of a hospital consisting of doctor, nurse, and patient objects. Each patient is modelled as a quadruple consisting of the patient's name, temperature, heart rate, and a value indicating his/her level of pain medication. Each patient is monitored by a single doctor and multiple nurses that must stay informed of the patient's vital signs. Based on the current readings, doctors and nurses can respond to queries regarding the health of patients under their care. Doctors can also adjust the level of pain medication prescribed to each patient.

The requirement that doctors and nurses stay informed of the current state of their patients calls for the use of the *Observer* pattern. The intent of the pattern is to keep a group of observer objects *consistent* with the state of a subject object. In this case, the observer *role* is played by doctor and nurse objects, and

```
1 public class Patient {
2   private String name; private int temp, hrtRt, medLvl;
3   private Set<Nurse> nurses; private Doctor doctor;
4   ...constructors, field accessor methods...
5   ...addNurse(n), removeNurse(n), setDoctor(d), unsetDoctor()...
6   public void checkVitals() { temp=...; hrtRt=...; notify();  }
7   public void adjustMeds(int newLvl) { medLvl = newLvl; }
8   private void notify() {...call update() on nurses and doctor...} ...
9 public class Nurse {
10  private HashMap<Patient,Integer> vitals; ...constructors...
11  public void update(Patient p) { vitals.put(p, p.getTemp()); }
12  public String getStatus(Patient p) {
13    int t = vitals.get(p);
14    if((t>90)&&(t<105)) return("good"); else return("bad"); } }
```

Fig. 1. Hospital Simulation Code (partial)

the subject role is played by patient objects. Key portions of the *Java* code for this system are shown in Figure 1. When applying the *Observer* pattern, system designers are guided by the pattern description presented in [1]. The style of presentation used in this catalog is common, and consists of an informal description of the problem, a discussion of the properties of the prescribed solution, and UML-like diagrams and code fragments that illustrate canonical applications. This type of description is useful in a number of ways. It is clear from the discussion in [1], for example, that a subject object should provide attach(o) and detach(o) methods for adding and removing an observer (o) from the set of objects *observing* its state. It is also clear that the subject should provide a notify() method that is invoked "whenever a change occurs that could make its observer's state inconsistent with its own." notify() should in turn invoke update() on each attached observer; update() will "reconcile its [the observer's] state with that of the subject." But how will a subject determine whether a change is significant enough to cause it to become inconsistent with its observers? Indeed, what does it mean to say that a subject's state is *inconsistent* with an observer? Similar questions arise when applying other patterns, and are not addressed by the informal descriptions. These are the types of ambiguities that can lead to software defects.

As an example, consider Patient.addNurse(). When this method is invoked, should notify() be called? After all, the execution of addNurse() modifies the state of the patient by adding a new nurse to the patient's set of attached observers. But this modification involves portions of the patient's state that are irrelevant from the point of view of the doctors and nurses already attached to the patient. Stated another way, the change is *insignificant*, and a call to notify() is unnecessary. Consider, however, the attaching nurse. Unless some action is taken, the nurse will not have information about the state of the patient when addNurse() finishes. Hence, if a patient query were issued to this nurse immediately following the

completion of addNurse(), the nurse might return random information about the patient! To prevent this, the addNurse() method must include a call to the update() method of the attaching nurse. This is a subtle issue that is not addressed in the informal description.

Consider a modification to the system. In the new system, nurses are responsible for monitoring vital signs *and* medication levels. The notion of *consistency* will naturally be revised to require that each nurse be aware of the current value of the patient's medLvl field. Designers will presumably revise Nurse.update() to save this information, and also revise Nurse.getStatus() to include the patient's medication level. But this is not sufficient! Changes made to a patient through adjustMeds() will not trigger calls to notify(). As a result, calls to adjustMeds() may leave nurses with inconsistent views of their patients. This is a surprisingly subtle bug given the simplicity of the system. With respect to the pattern, the only change dictated by the new requirements seems to be a redefinition of what it means for the state of a nurse to be *consistent* with the state of a patient — and the corresponding changes in Nurse.update() and Nurse.getStatus(). But as we have seen, this is inaccurate. The change in the notion of *consistency* demands a corresponding change in the notion of *significant change*. More precisely, a change in medLvl is now significant, and should therefore trigger a call to notify(). In general, the *concepts* used in describing a pattern must often satisfy relationships that are not clear from the informal descriptions. Our contracts are designed to make these conditions clear to designers and implementers.

The types of ambiguities that lead to system defects in our hospital simulation are the same types of defects that lead to failures in actual systems. Our pattern specifications are designed to eliminate these ambiguities, while retaining the flexibility present in the informal descriptions. In the event that an implementation error is introduced, our monitoring tools are designed to detect the error before the system is deployed.

3 Design Pattern Contracts

The partial grammar of our contract language is shown in Figure 2. A contract begins with a declaration of the *auxiliary concepts* used throughout its body ($\langle concepts \rangle$). Each specifies a relation involving one or more states of the objects that play *roles* in the pattern being specified. Their purpose is to capture points of variation that occur across different applications of the pattern. Each includes a concept identifier ($\langle coId \rangle$) and the list of roles over which the concept is defined ($\langle rIds \rangle$). The Observer contract, for example, declares the concept *Consistent*(Subject, Observer) to capture the notion of *consistency* between a subject and an observer. Since the meaning of consistency varies from one system built using the *Observer* pattern to another, the contract defers the definition of *Consistent*() to the *subcontract* corresponding to a particular application. By expressing our contracts in terms of auxiliary concepts, but deferring their definitions to subcontracts, we achieve descriptive precision without compromising pattern flexibility. As we saw, however, *arbitrary* flexibility should not be allowed;

```
1  ⟨contract⟩       → pattern contract ⟨pId⟩ {
2     ⟨conceptBlock⟩ ⟨instantiation⟩ ⟨invariant⟩ ⟨roleContracts⟩ }
3  ⟨conceptBlock⟩   → concepts: ⟨concepts⟩ ⟨constraints⟩
4  ⟨concept⟩        → ⟨coId⟩(⟨rIds⟩);
5  ⟨constraints⟩    → constraints: ...predicate on auxiliary concepts...
6  ⟨instantiation⟩  → instantiation: ⟨rId⟩.⟨mId⟩(⟨args⟩) { ⟨cond⟩ };
7                       lead: (target|source|⟨arg⟩|...code...);
8  ⟨invariant⟩      → invariant: ...assertion on roles and concepts...
9  ⟨roleContract⟩   → [lead] role contract ⟨rId⟩ {
10    ⟨fields⟩ ⟨methods⟩ ⟨others⟩ ⟨enrollment⟩ ⟨disenrollment⟩ }
11 ⟨field⟩          → ...role field declaration...
12 ⟨method⟩         → ...standard method specification...
13 ⟨others⟩         → others: ...standard method specification...
14 ⟨enrollment⟩     → ...analogous to instantiation...
15                      enrollee: (target|source|⟨arg⟩|...code...);
16 ⟨disenrollment⟩  → ...analogous to instantiation...
```

Fig. 2. Grammar of Pattern Contracts (partial)

the concept definitions corresponding to a particular system must often satisfy conditions to ensure that the intent of the pattern is not violated. Hence, the pattern contract also specifies *constraints* that must be satisfied by the concept definitions supplied in any subcontract (⟨*constraints*⟩).

The next element specifies the conditions that must be satisfied to *instantiate* a new instance of the pattern (⟨*instantiation*⟩). Pattern instantiation is associated with the invocation of a particular role method or constructor specified in the pattern contract (⟨*rId*⟩.⟨*mId*⟩(⟨*args*⟩)). The contract specifies any state conditions that must be satisfied upon termination of the method, as well as the object that will serve as the *lead object* of the newly created pattern instance. The lead object serves as a handle to refer to its corresponding pattern instance in other portions of the contract. The lead object may be specified as the target of the invocation (target), the source of the invocation (source), one of the arguments to the invocation (⟨*arg*⟩), or some other object specified using a code fragment.

The next element specifies a *pattern invariant* that captures the behavioral guarantees that should be expected if the contract requirements are satisfied (⟨*invariant*⟩). These properties are expressed using an assertion involving the objects enrolled in the pattern instance, and the auxiliary concepts defined by the contract. This assertion will be satisfied whenever control is outside of the participating objects. In effect, this portion of the contract captures the "defined properties" discussed in [2]: the system behaviors that result when the pattern is used correctly.

The final portion consists of *role contracts* that specify the requirements associated with objects *enrolled* to participate in the pattern (⟨*roleContracts*⟩). One of these roles will be flagged as the *lead role*, indicating that its instances may serve as lead objects. The Observer contract, for example, specifies Subject and Observer role contracts, corresponding to the two types of objects that participate in the pattern. The Subject role is flagged as the lead role. Each role contract

begins by specifying the *role fields* to which an object's state must be mapped when it plays the corresponding role (⟨*fields*⟩). Similarly, it specifies the *role methods* that an enrolled object must provide, given the appropriate interface mappings in a subcontract, including pre- and post-condition specifications of the method behaviors (⟨*methods*⟩). These specifications are expressed in terms of role fields and auxiliary concepts. In addition, since an enrolled object may provide methods that do not correspond to any of the role methods, the role contract specifies conditions that prevent these *other* methods from violating the intent of the pattern (⟨*others*⟩). Finally, the role contract specifies the conditions that must be satisfied for an object to *enroll* or *disenroll* (⟨*enrollment*⟩,⟨*disenrollment*⟩). These clauses are defined analogously to the pattern instantiation clause. The only difference in the enrollment clause is that in addition to specifying the lead object (to identify the pattern instance into which the object will enroll), it specifies the enrolling object. The disenrollment clause is analogous.

3.1 Special Notations

Before turning to an example, there are two special notations used in our pattern contracts and subcontracts that are important to consider. The first is the keyword players, used to denote the *sequence* of *player* objects enrolled in a pattern instance. The order of the objects within the sequence corresponds to the order in which the objects enrolled. We use indexing notation to refer to a particular object or subsequence of objects. players[0], for example, refers to the first enrolled object, and players[1:] refers to the subsequence of enrolled objects beginning at the second object.

The second notation allows us to impose conditions on the method calls made by a method during its execution. Addressing such requirements is important since many patterns call for particular methods to be invoked under various conditions. To achieve this, we use the notion of a *call sequence* (or "*trace*"), and use the symbol τ to denote the call sequence associated with a method invocation. Each element within τ represents a method call, and records (*i*) the name of the method invoked, (*ii*) the target of the invocation, and (*iii*) any arguments to the call. We use *dot* notation to denote the projection associated with calls to particular methods of particular objects. τ.o.m, for example, represents the subsequence of calls to method m() on object o. $|\tau|$ denotes the number of calls recorded in the call sequence τ.

3.2 The Observer Contract

Consider the partial contract for the *Observer* pattern shown in Figure 3. The contract declares the auxiliary concepts *Consistent*() and *Modified*(). As explained earlier, *Consistent*() captures the notion of *consistency* between a subject and an observer. *Modified*() captures the notion of *significant change* within a subject. The latter concept is later used to express the requirement that every significant change within a subject result in a call to notify(). The former concept is used to require that Observer.update() appropriately update the observer's state. The constraint imposed on these concepts requires that if a subject's state changes

```
pattern contract Observer {
  concepts:
    Consistent(Subject,Observer); Modified(Subject,Subject);
  constraints: ∀s1,s2,o1:::(¬Modified(s1,s2) ∧ Consistent(s1,o1))
                                  ⇒ Consistent(s2,o1)
  instantiation: Subject.Subject() { obs=∅ }; lead: target;
  invariant: Subject(players[0]) ∧ Observer(players[1:]) ∧ ... ∧
             ∀ob:ob ∈ players[1:]::Consistent(players[0],ob)
```

Fig. 3. Observer Pattern Contract (partial)

from s1 to s2, and the change is deemed *insignificant*, then any observer state consistent with s1 must also be consistent with s2. This constraint prevents the types of incompatible concept definitions that lead to software defects in our hospital system. More precisely, it prevents definitions of *Modified*() and *Consistent*() that would allow a subject to omit a call to notify() after a change that could lead to *inconsistency* with one or more of its observers.

The instantiation clause specifies that a new instance of the pattern is created each time a new subject object is created. Further, it requires that at the point of instantiation, the subject's obs set be empty (since no observers have yet enrolled). Finally, it states that the newly created subject will serve as the *lead* object of the pattern instance.

The invariant clause captures the intent of the pattern, the "defined properties" that may be expected if the contract requirements are met. It states that the first object to enroll in a pattern instance will play the role of Subject and all other enrolled objects will play the role of Observer. Most important, it states that whenever control is outside of the participating objects, all of the enrolled observers will be in states that are consistent with the current state of the subject.

The partial Subject role contract is shown in Figure 4, and specifies the state components and method behaviors that must be provided by objects playing the Subject role. To benefit from the *pattern invariant*, these requirements must

```
lead role contract Subject {
  Set<Observer> obs;
  void attach(Observer ob):
    pre: ob ∉ obs
    post: (ob=#ob) ∧ ¬Modified(#this,this) ∧(obs=(#obs∪{ob}))
          ∧ (|τ|=1) ∧ (|τ.ob.update|=1)   ...detach(ob)...
  void notify():
    post: (obs=#obs) ∧ ¬Modified(#this,this) ∧ (|τ|=|obs|) ∧
          ∀ob:ob ∈ obs::(|τ.ob.update|=1)
  others:
    post: (obs=#obs) ∧ ((¬Modified(#this,this) ∧ (|τ|=0)) ∨
          (|τ.this.notify|=1))
```

Fig. 4. Subject Role Contract (partial)

be satisfied under the field and interface mappings specified in the relevant subcontract. (We will discuss these mappings shortly.) The role contract states that each subject must provide a Set component, which will be used to store the set of attached observers. It also includes specifications for the attach(), detach(), and notify() methods. In the post-conditions of these methods, we use the # notation to denote the pre-conditional value of an object. Hence, the attach() method is required to preserve the reference to the attaching observer (ob), to leave the subject *unmodified*, and to add the attaching object to the set of attached observers (obs). Further, the call sequence conditions require that update() be invoked on the attaching object. This requirement guarantees — given the specification of Observer.update() (omitted) — that the observer will be in a state that is *consistent* with the current state of the subject when attach() terminates. Again, this condition is important to prevent the types of inconsistency defects encountered in our hospital system. detach() is defined analogously, but omits the call sequence conditions. The final method, notify(), is required to preserve the set of attached observers, and to leave the subject *unmodified*. The call sequence conditions require that the method invoke update() on each attached observer.

The others clause imposes requirements on the methods provided by a player beyond those that map to attach(), detach(), and notify(). All of these *other* methods are required to preserve the set of attached observers. Further, if one of these methods makes a *significant change* in the subject (*i.e.*, *Modified*(#this, this) is *true*), it must include a call to notify(). As we have seen, this method will in turn invoke update() on each attached observer, ensuring their *consistency* with the new state of the subject.

The Observer role contract is defined in the same manner as the Subject role contract. observer objects are required to maintain a reference to their subject, and to provide an update() method. The post-condition of update() requires that it leave the observer in a state that is *consistent* with the current state of the observer's subject. The others clause imposes similar requirements to ensure that the pattern invariant is respected.

3.3 Pattern Subcontracts

A subcontract *specializes* a pattern contract for use, customizing its requirements and behavioral guarantees to the needs of a particular system. This specialization mechanism is essential for preserving the *flexibility* of our pattern contracts. The partial grammar of our subcontract language is shown in Figure 5. Each subcontract begins by specifying a set of *role maps* that characterize the manner in which particular system classes can be viewed as their corresponding role types (⟨*roleMaps*⟩). The Hospital subcontract, for example, defines role maps that allow us to view a patient as a subject, a nurse as an observer, and a doctor as an observer. Each role map consists of a *state map* and an *interface map* (⟨*stateMap*⟩,⟨*interfaceMap*⟩). A state map defines functions that map an object's fields to the fields defined by its role (⟨*rfId*⟩). These functions are written in the form of code fragments to simplify the expression of the mappings, and to simplify the task of generating the appropriate monitoring code. Similarly, an

```
1  ⟨subcontract⟩  → subcontract ⟨sId⟩ specializes ⟨pId⟩ {
2    ⟨roleMaps⟩ ⟨concDefBlock⟩ }
3  ⟨roleMap⟩      → rolemap ⟨cId⟩ as ⟨rId⟩ {
4    ⟨stateMap⟩ ⟨interfaceMap⟩ }
5  ⟨stateMap⟩     → state: ⟨fieldMaps⟩
6  ⟨fieldMap⟩     → ⟨rfId⟩ = {...code...}
7  ⟨interfaceMap⟩ → methods: ⟨methodMaps⟩
8  ⟨methodMap⟩    → ⟨rmId⟩(⟨rmArgs⟩):⟨classMethods⟩
9  ⟨classMethod⟩  → ⟨cmId⟩(⟨cmArgs⟩)[{⟨argMaps⟩}]
10 ⟨concDefBlock⟩ → auxiliary concepts: ⟨concDefs⟩
11 ⟨concDef⟩      → ⟨coId⟩(⟨coArgs⟩) {...code...}
```

Fig. 5. Grammar of Pattern Subcontracts (partial)

interface map specifies mappings from the class methods and arguments to their corresponding role methods and arguments (⟨*methodMaps*⟩,⟨*argMaps*⟩). As we will see, multiple class methods may be mapped to a single role method.

The final element of a subcontract provides *auxiliary concept* definitions appropriate to the given system (⟨*concDefs*⟩). Each auxiliary concept is written as a code fragment expressed over the classes mapped to the concept arguments. Each concept returns a boolean value indicating whether the relation is satisfied given the states of the objects passed as argument. When the auxiliary concept definitions and role maps are substituted into the contract being specialized, the resulting specification characterizes the pattern requirements and behavioral guarantees specific to the system in question.

3.4 The Hospital Subcontract

As an example, consider the partial subcontract for our hospital system shown in Figure 6. The subcontract begins by defining the role map that allows us to view a patient as a subject. Under this view, the state map specifies that the subject's obs field is realized as the set containing all of the elements in nurses, plus the object referenced by doc, if any. The interface map specifies that both addNurse(n) and setDoctor(d) play the part of attach(o). In both cases, the argument to the class method corresponds directly to the argument to the role method. removeNurse(n) and unsetDoctor() are defined analogously, except that in the case of unsetDoctor(), which takes no arguments, the argument to detach(ob) is played by the patient's doc field. Patient.notify() corresponds directly to Subject.notify(). The Nurse as Observer, and Doctor as Observer role maps are defined in a similar manner.

The definition of *Modified*() specifies that any change in patient.temp or patient.hrtRt is considered a *significant change*. The two definitions of *Consistent*() are more interesting. Since each nurse and each doctor may be involved in multiple pattern instances, each may store information about multiple patients. Or more generally, each observer may store information about multiple subjects. In

```
1  subcontract Hospital specializes Observer {
2    rolemap Patient as Subject {
3      state: obs = { Set<Observer> obs =
4                       new HashSet<Observer>(nurses);
5                     if(doc!=null) obs.add(doc); return(obs); }
6      methods:
7        attach(Observer ob):addNurse(ob),setDoctor(ob)
8        detach(Observer ob):removeNurse(ob),
9                            unsetDoctor(){ob=doc}   ...notify()...
10   ...Nurse/Doctor as Observer rolemaps...
11   auxiliary concepts:
12     Modified(Patient p1, Patient p2) {
13       return((p1.temp!=p2.temp) || (p1.hrtRt!=p2.hrtRt)); }
14     Consistent(Patient p, Nurse n) {
15       return(p.hrtRt == n.vitals.get(lead)); }
16     ...Consistent(Patient, Doctor) concept definition...
```

Fig. 6. Hospital Subcontract (partial)

reasoning about a particular pattern instance, it must be possible to project out those portions of an observer's state relevant to the pattern instance (and therefore the subject) in question. This is achieved using the lead object (a surrogate pattern instance identifier) as an index into the observer's state. The lead keyword refers to the lead object in a way that is analogous to the use of the this keyword in object-oriented languages. Hence, in the definition of *Consistent*() corresponding to nurse objects, the lead object is used to retrieve the patient information corresponding to the pattern instance in question. The case involving doctor objects is analogous.

4 Pattern Contract Monitors

In addition to having pattern contracts that are both precise and flexible, it is important to have supporting software tools that can assist in determining whether the requirements specified by a contract are satisfied. To achieve this, we have developed a monitor generation tool based on our pattern contract language. Given the pattern contracts and subcontracts underlying a particular system design, our tool generates runtime monitoring code that signals any violations of the contract requirements. Given that the assertions to be checked are crosscutting, we chose to use an *aspect-oriented* approach. Our current implementation targets Java-based systems, and generates aspects in *AspectJ* [4][1]. The monitor generation process is illustrated in Figure 7.

The monitoring code produced for a given contract/subcontract pair consists of one *abstract aspect* and one *concrete subaspect*. The abstract aspect contains

[1] The tool, including source code, documentation, and system examples, is available for download at: *http://www.cse.ohio-state.edu/~tyler/MonGen/*

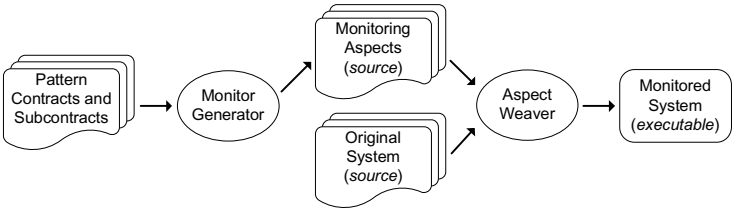

Fig. 7. Monitor Generation Process

checking logic common across all specializations of the contract, and the subaspect tailors this logic to the particular specialization specified by the subcontract. Consider, for example, the abstract aspect generated from the Observer contract (Figures 3 and 4) shown in Figure 8. The aspect begins by declaring interfaces for each of the roles defined in the pattern contract (Line 2). These interfaces are mapped to the appropriate system classes in the subaspect based on the role maps included in the subcontract. The subaspect generated from the Hospital subcontract (omitted), for example, maps the Subject interface to the Patient class using *AspectJ*'s declare parents construct. This effectively forces the Patient class to implement the (empty) Subject interface. Similar mappings are defined for Nurse and Doctor. This allows methods defined in the abstract aspect, which are defined in terms of Subject and Observer objects, to work with Patient, Nurse, and Doctor objects.

The aspect next defines state components required to monitor multiple pattern instances (Lines 3–4). The first of these components is a *pattern instance map* (instanceMap) that maintains a mapping from each lead object to its corresponding PatternInstance object. Each PatternInstance stores information about a single pattern instance, including references to the enrolled objects and the roles that these objects play. This information is required to check certain instantiation, enrollment, and disenrollment conditions — such as those that make use of the players keyword. The instance map is updated when a pattern instance is created or destroyed, and when an object enrolls or disenrolls.

The second state component is the *trace stack* (traces), which stores call sequence (τ) information about each of the active role methods. This information is required to check the call sequence conditions specified in the pattern contract. The trace stack is updated before and after every role method invocation.

The aspect next declares *pointcuts* corresponding to each of the role methods specified in the pattern contract (Lines 5–6). The *advice* bound to these pointcuts is responsible for checking the appropriate role method requirements, as well as for updating the pattern instance map and trace stack. Since, however, the mapping between class methods and role methods varies from application to application, the pointcuts are declared *abstract*. Pointcut definitions are supplied in the subaspect based on the interface maps specified in the relevant subcontract. The subaspect generated from the Hospital subcontract, for example, maps the sub_attach() pointcut (corresponding to Subject.attach()) to the execution of either Patient.addNurse() or Patient.setDoctor(). Similar pointcuts are used to capture pattern instantiation, object enrollment, and disenrollment. Pointcuts

```
1  public abstract privileged aspect ObserverM {
2    interface Subject{} interface Observer{}
3    private HashMap<Subject,PatternInstance> instanceMap;
4    private TraceStack traces;
5    ...pointcuts for role constructors, role methods, and other methods:
6    abstract pointcut sub_attach(Subject _this, Observer ob); ...
7    ...auxiliary concept methods:
8    public abstract boolean Modified(Subject a1, Subject a2);
9    public abstract boolean Consistent(Subject a1, Observer a2);
10   ...role state accessor methods:
11   public abstract Set<Observer> sub_obs(Subject _this);
12   public abstract Subject obs_sub(Observer _this,Subject lead);
13   ...assertion checking / bookkeeping advices:
14   after(Subject _this, Observer ob): sub_attach(_this, ob){
15     ...get #ob, #this from caller trace record...
16     assert((ob==pre_ob) && !Modified(pre_this, _this) &&
17       sub_obs(_this).containsAll(sub_obs(pre_this)) && ... &&
18       (traces.current().length() == 1) &&
19       (traces.current().limit(ob,"update").length() == 1));
20     ...update caller trace record... } ... }
```

Fig. 8. The ObserverM Contract Monitor (partial)

are also declared to capture the *other* methods of the class(es) mapped to each role. These pointcuts are defined to include all of the class methods *except* those bound to role methods.

Recall that the requirements specified in the pattern contract are expressed in terms of auxiliary concepts and role fields. Since the realizations of these elements vary, they are captured using abstract methods, deferring their definitions to a subaspect. ObserverM, for example, declares Modified() and Consistent() methods corresponding to the auxiliary concepts of the same name (Lines 7–9). It also declares abstract methods corresponding to Subject.obs and Observer.sub (Lines 10–12). Each of the latter methods returns the appropriate role field value when the argument passed as input is viewed as an instance of its role. The implementations of the auxiliary concept and role field methods are supplied in the subaspect based on the concept definitions and state maps provided in the relevant subcontract. Since these elements are defined (in the subcontract) in terms of code fragments, the code generation task is straightforward.

Note that for Observer.sub, the corresponding method takes an additional argument. Since the Observer role is not flagged as lead in the pattern contract, each observer may participate in multiple pattern instances. This means that each observer (conceptually) stores multiple copies of the sub field — one copy corresponding to each pattern instance. The lead argument is used to identify the pattern instance under which the state mapping should be performed.

The final portion of the aspect defines the advice bound to each pointcut. The checking code within the advice is generated based on the assertions specified in the pattern contract. The before and after advice bound to each pointcut

	Size of Contract		Size of Subcontract		Execution Time (in ms)	
	Specific.	Abs. Aspect	Specific.	Subaspect	w/ Montr.	w/o Montr.
Observer	1723	14,701	866	3967	2657	172
Memento	907	11,116	771	3730	3610	391
Chain of Resp.	592	5658	377	1845	2453	297

Fig. 9. Code Size and Runtime Overhead. [Pentium-IV @ 2.53GHz, 512MB RAM, Windows XP Pro SP 2, Sun JVM 1.5.0_04].

is responsible for checking the relevant pre- and post-conditions, respectively. The advice is also responsible for updating the pattern instance map and the trace stack. A portion of the after advice generated from the specification of Subject.Attach() is shown in the figure (Lines 14–20). The advice bound to the remaining pointcuts is defined in a similar manner.

One difference between the advice bound to Subject methods and the advice bound to Observer methods is that the latter begins by identifying *every* pattern instance in which an observer participates. The relevant assertions are then checked in the context of each pattern instance. The corresponding lead object is retrieved from the pattern instance map, and serves as the second argument when invoking the role field method corresponding to Observer.sub.

4.1 Code Size and Runtime Overhead

We have applied our approach to several different patterns and systems. Space restrictions preclude a detailed discussion of the results, but it is interesting to consider the gross relationship between contract/subcontract size and the size of the corresponding monitoring code. It is also interesting to consider the runtime overhead introduced when this code is woven into an actual system. Figure 9 presents the data corresponding to the use of our contracts for *Observer*, *Memento*, and *Chain of Responsibility* when used in monitoring the canonical system examples presented in [1]. As a gross estimate, we measure size in terms of non-whitespace characters. We emphasize that this is a preliminary analysis.

5 Related Work

We are not the first to consider pattern formalization. Eden *et al.* [5], for example, propose a higher-order logic formalism that captures patterns as formulae. Each formula consists of a declaration of the participating classes, methods, and inheritance hierarchies, and a conjunctive statement of the relations among them. While rich structural properties can be expressed, there is limited support for capturing behavioral properties. The formalism does not, for example, provide constructs for referring to pre- and post-conditional values, nor does it provide a concept analogous to our method call sequences. By contrast, Mikkonen's work [6] focuses almost exclusively on behavioral properties. In his approach, patterns are specified using an action system notation. Data classes model pattern participants, and guarded actions model their interactions. The approach

is well-suited to reasoning about temporal properties. One limitation, however, is that the separation of actions and data is structurally inconsistent with the OO paradigm, making it difficult to express most structural properties. Further, Mikkonen's specifications cannot be specialized to the needs of particular systems; thus pattern flexibility may be seriously compromised.

Helm *et al.* [7] describe a contract formalism that shares similarities with ours. For example, their formalism includes a construct similar to our auxiliary concepts. It does not, however, provide a way to impose constraints that would prevent definitions of these concepts from violating a pattern's intent. The formalism also includes support for specifying the relative order of method invocations, but the support is limited. It is impossible, for example, to quantify over a method call sequence to require that a particular method be invoked exactly once, or alternatively, that a particular method not be invoked at all. Finally, there is nothing analogous to our use of the others clause to prevent non-role methods from violating a pattern's intent.

In [8] and [9], we describe principles of pattern formalization and runtime monitoring, but do not consider a general pattern specification language, pattern specializations, or automated monitor generation. We provide an overview of the specification and monitoring approach in [10], but do not go into the technical detail presented here. For example, [10] presents only a subset of the specification language; the subset cannot, for example, accommodate multiple pattern instances. Other important contributions presented here that are absent from [10] include a detailed system and subcontract example, a presentation of the generated monitoring code, an analysis of the code size and runtime overhead associated with monitoring, and a discussion of how the approach supports a pattern-centric software lifecycle (Section 6).

Runtime assertion monitoring of OO systems has a long history [11–13], and some authors have considered aspect-based approaches. Lippert and Lopes [14] use *AspectJ* to refactor pre- and post-conditional assertion checking code. Gibbs and Malloy [15] propose using aspects to monitor class invariants involving temporal properties. To our knowledge, however, we are the first to investigate contract monitors for design patterns.

6 Discussion

Our work was motivated by three observations. First, informal pattern descriptions leave a potential for ambiguity and misunderstanding that jeopardizes the correct use of patterns. Second, there is limited tool support to assist in identifying pattern implementation errors. Third, as a system evolves, its *design integrity* may erode under maintenance; it may no longer remain faithful to the patterns underlying its design. We presented two contributions to address these problems. The first was a formalism for expressing *pattern contracts* that capture the implementation requirements and behavioral guarantees associated with a range of patterns. The formalism includes support for *subcontracts* that capture

the ways in which patterns are specialized for use in particular systems. Thus, we are able to specify properties common across all applications of a pattern, while accommodating the inherent variation that occurs across those applications. We illustrated the approach by developing the contract for the *Observer* pattern, and a corresponding subcontract for a simple system built using this pattern.

Our second contribution was a *monitor generation tool*. Given the pattern contracts and subcontracts underlying a system design, our tool produces a set of aspects in *AspectJ* that monitor the system's runtime behavior to check whether the contract requirements are violated. We presented some of the key details concerning the aspects generated by the tool, as well as the structure of the tool itself. Finally, we presented preliminary figures to show the code and runtime overhead involved in using the tool to monitor a system during its execution.

These contributions, along with our planned extensions, provide the basis for a *pattern-centric* software lifecycle. At the foundation of the lifecycle is a *contract catalog* that complements existing pattern catalogs. The catalog is an evolving document that we plan to make accessible through the web. We hope that researchers interested in lightweight formal methods will contribute to its development. Community involvement is essential in ensuring that the contracts faithfully capture the intent of the patterns specified. Members of a design team will be able to consult the catalog to ensure a common understanding of the requirements associated with the patterns underlying a particular design.

As the design and implementation details of the system are fleshed out, part of the design team will be charged with creating the corresponding subcontracts. In addition to guiding the implementation, the subcontracts will allow implementation and maintenance teams to generate appropriate runtime monitoring code. Executing this code will enable the team to identify pattern implementation errors more easily — from early implementation through evolution.

Note that while developing a pattern contract requires reasonable facility with formal notations, developing a subcontract is a task that will likely appeal to system developers. Indeed, this is one reason why this portion of the formalism resembles a programming notation more than it resembles formal mathematics. As part of our future work, we plan to assess the degree of effort involved in developing and maintaining these subcontracts. This will allow us to perform a cost-benefit analysis by comparing this effort to the benefit received when using the approach. We also plan to investigate techniques for generating test suites that ensure suitable *coverage* of the patterns' used in a system.

Another exciting possibility is a *pattern-centric visualization tool*. During a system's execution, the monitoring code will save appropriate information relevant to the patterns used in the system. The visualization tool will then take this information and play it back in the form of a "*slow-motion-video*", allowing the user to go back and forth in the system's execution, focusing on the interactions among groups of objects interacting according to the patterns of interest. This will be of particular value to new members of a design team since it will enable them to quickly develop a pattern-centric understanding of relevant systems.

References

1. Gamma, E., *et al.*: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
2. Buschmann, F., *et al.*: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons, Inc. (1996)
3. Schmidt, D., *et al.*: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. John Wiley & Sons, Inc. (1996)
4. Kiczales, G., *et al.*: An overview of AspectJ. In: ECOOP. (2001) 327–353
5. Eden, A.: Formal specification of object-oriented design. In: CSME-MDE. (2001)
6. Mikkonen, T.: Formalizing design patterns. In: ICSE, IEEE (1998) 115–124
7. Helm, R., *et al.*: Contracts: Specifying behavioral compositions in object-oriented systems. In: OOPSLA/ECOOP, ACM (1990) 169–180
8. Soundarajan, N., Hallstrom, J.: Responsibilities and rewards: Specifying design patterns. In: ICSE. (2004) 666–675
9. Soundarajan, N., *et al.*: Specifying and monitoring design pattern contracts. In: SAVCBS/ICSE. (2004) 87–94
10. Tyler, B., *et al.*: Automated generation of monitors for pattern contracts. In: SAC. (2006) (*to appear*).
11. Meyer, B.: Object-Oriented Software Construction. Prentice Hall (1988)
12. Rosenblum, D.: A practical approach to programming with assertions. IEEE TSE **21** (1995) 19–31
13. Burdy *et al.*, L.: An overview of JML tools and applications. STTT (2005) (to appear).
14. Lippert, M., Lopes, C.: A study on exception detection and handling using aspect-oriented programming. In: ICSE. (2000) 418–427
15. Gibbs, T., Malloy, B.: Weaving aspects into C++ applications for validation of temporal invariants. In: CSMR. (2003) 249–258