Reasoning About Design Patterns: A Case Study

Jason O. Hallstrom and Neelam Soundarajan
Computer Science and Engineering
Ohio State University, Columbus, OH 43210
e-mail: {hallstro, neelam}@cis.ohio-state.edu

Abstract

Design patterns are valuable both for designing and for documenting software systems. Patterns are usually described informally. While informal descriptions are very useful, in order to be sure that designers have a precise understanding of the requirements that must be met when applying a given pattern, and to be able to reliably predict the behaviors that systems built using specific patterns will exhibit, we also need precise specifications of the patterns.

In this paper, we apply an approach to formally specifying patterns [14] to the Memento pattern as a case study. One important aspect of patterns is their flexibility. Our case study shows that this flexibility is not compromised by our formalization.

Keywords: Software design, Design patterns, Software requirements, Formal methods.

1 Introduction

Design patterns [1, 2, 8, 9, 13] have, over the last decade, fundamentally changed the way we think about the design of large software systems. As Buschmann *et al.* [2] put it, "patterns support the construction of software with defined properties". But to fully realize these benefits, we must have precise ways to *reason* about these design patterns that we can use to ensure that the patterns are being applied in the ways they are intended to be. Our goal in this paper is to apply an approach to reasoning about patterns that we have developed, a preliminary version of which was presented in [14], to the Memento pattern as a practical case-study.

When reasoning about a pattern, we have to address two distinct aspects. The first, which we call the *responsibilities* component of the specification of a pattern, will consist of the conditions that a designer adopting the pattern must ensure are satisfied with regard to the structure of the classes, the behaviors of particular methods of the classes, and the interactions between various classes of the system. The second, which we call the *rewards* component, will specify the particular behaviors that the resulting system is guaranteed to exhibit, the 'defined properties' of [2]; but this guarantee applies only if all the requirements contained in the responsibilities component are satisfied.

Consider the Memento pattern [8] which will serve as our case-study. The purpose of this pattern is to allow an object (the 'Caretaker') to take a *snapshot* of the state of another object (the 'Originator') without breaking the encapsulation of the Originator, so that at a later point the Caretaker can, if necessary, restore the state of the Originator; the snapshot is the 'Memento'. This is represented in the standard UML representation of the pattern in Fig. 1.

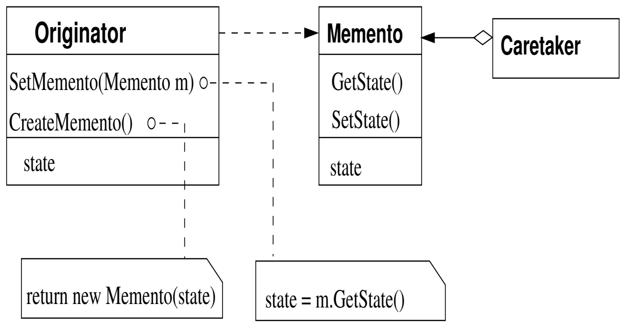

Figure 1. Memento Pattern

For example, a particular object o1 might at some point want to save the state of another object o2, and hence take a snapshot of o2's current state; at this point, o1 is the Caretaker, and o2 the Originator. At a later point in the execution, a third object o3 might want to save the state of o1, and hence take o1's snapshot; at this point, o1 is the Originator and o3 the Caretaker. Thus the same object o1 is playing the roles of Caretaker and Originator at different times. Hence a pattern is made up of *roles*; there are three roles in this pattern: Originator, Caretaker, and Memento. Individual objects will, at runtime, enroll in particular roles.

Fig. 1 suggests, given the body of Originator.CreateMemento(), that the Memento is (or contains) a copy of the *entire* state of the Originator; and, given the body of SetMemento(), that when we restore the state, we restore the entire Originator state back from the Memento. While this is one way to implement the pattern, it is often not the best approach since, depending on the size of the Originator state, and the number of Mementos created, it may be expensive or inappropriate. The commentary associated with the pattern in [8] notes that the Memento may store as little of the Originator state 'as necessary at the Originator's discretion'; but how does a designer determine exactly what is necessary? The description goes on to say that the information saved must be sufficient to 'restore the internal state of the Originator', but again, what precisely does that mean? That every byte of that state be restored by SetMemento()? It is such questions that our formal specifications will help answer.

It has occasionally been suggested that the real key to reuse is *design reuse*, as embodied in patterns; an approach such as ours should also allow us to reuse the reasoning needed to understand systems built using particular patterns. But there is an inherent risk in formalizing patterns in that *flexibility*, a key aspect of many patterns, may be lost. Thus in the case of Memento, if we adopt one definition for what it means for the Originator to be reset to an 'appropriate state', the pattern may not be usable when we have a different notion of this concept. Avoiding this risk was one of our goals; our approach allows us to characterize a pattern precisely while also retaining its flexibility.

The rest of the paper is organized as follows: In Section 2, we develop the basic ideas of our approach. In Section 3, we apply our approach to the Memento pattern. In Section 4 we consider related work. In the final section, we summarize our approach to formalizing patterns, and consider pointers for future work.

2 Roles, Auxiliary Concepts, and Constraints

Consider again the Memento pattern. At a particular point in the execution of our system, we may have a group of objects interacting according to the Memento pattern. That is, one object is playing the Originator role, another the Caretaker role, a third object (or possibly more than one if the caretaker object[1] has taken multiple snapshots of the originator object) the Memento role. At the *same* time, another group of objects may also be interacting according to the Memento pattern; indeed, as in the case of object o1 in the scenario in Sec. 1, one or more objects might belong to both groups. To allow for this, we will say that there may be zero or more *instances* of a given pattern at any time during the execution of the system. Individual objects will *enroll* in a particular instance of a pattern to play a particular role; the same object may be enrolled simultaneously to play roles in different instances of the same or different patterns; but it *cannot* simultaneously play multiple roles in a given pattern instance.

How do we specify a pattern? Since we are dealing with roles not classes, there are no specific data members in terms of whose values we can express the *requires* and *ensures* clauses of the individual methods. Hence we introduce, in our pattern specifications, the needed data members as part of the individual roles of the pattern, and use these to specify these clauses. At the point where an object enrolls to play this particular role, the designer will have to specify a suitable mapping between the members of the object's class lcC and the methods and data members (that we introduced) of the role, and show that under this mapping the *requires* and *ensures* clauses of the role's methods are satisfied. As we will see, we may also need to refer to the other data members of C, for which we will use the notation αs (to denote "application-level state").

In addition to data members, we often also need to introduce specific *relations* involving two or more states of a role or between the states of different roles of the pattern. For example, consider again the question of what happens when the state of the originator is 'restored'. The question was whether this means that every bit and byte of the object has to be restored. While that may be appropriate in some applications, the Memento pattern does not demand it. For example, suppose we are dealing with graphics objects and use snapshots (the memento objects) that are lossy *jpeg* compressions of the graphics objects. When restored from such a memento, the restoration cannot have 100% fidelity. To allow for such cases, we will introduce a relation, Restored(s1, s2) of two states s1 and s2 of the graphics entity, this relation being satisfied if s2 is 'sufficiently similar' to s1, given the lossy nature of the jpeg memento involved. As we will see, the auxiliary concepts are the key to specifying the pattern precisely while at the same time retaining, indeed even *adding to*, its flexibility.

3 Specifying the Memento Pattern

For convenience, we split the specification of the Memento pattern into three pieces. The first, Fig. 2, is concerned with the pattern-level portion of the specification; Figs. 3 and 4 specify the individual roles. But this split is only for purposes of presentation; logically, it is inappropriate to split the specification in this manner since the main point of the pattern is to focus on how the roles interact with each other, so it does not make much sense to consider them separately from each other.

One interesting question that does not seem to have been addressed elsewhere came up as we built our specification. Suppose at time t2, we restore the state of an originator o using a memento m created at an earlier time t1. What should we do with the mementos (for the same originator) created between t1 and t2? Should we consider them *defunct* since o has been restored to an earlier point than the one at which these other mementos were created? In the specification below, in order to keep the presentation simple, we have ignored this issue. Near the end of the section, we consider how to modify the specification to disallow the use of defunct mementos.

[1]Names of roles will start with uppercase letters; the corresponding names starting with lowercase letters will refer to the object(s) playing the particular role(s).

```
pattern Memento {
  roles: Originator, Memento*, Caretaker*;
                                    //see note 1 below
  state:                                      // note 2
    Originator: null;
    Memento: OrigState orS;
    Caretaker: null;
  auxiliaryConcepts:                          // note 3
    relation: MemCopy(Orig.αs, Mem.αs);
    relation: SameMem(Mem.αs1, Mem.αs2):
                    reflexive, transitive;
    relation: StateInfo(orSI, Mem.αs);
    relation: Reset(Orig.αs1, Mem.αs, Orig.αs2);
    relation: Restored(Orig.αs1, Orig.αs2):
                    reflexive;                // note 4
  constraints:                                // note 5
    [MemCopy(os1,ms1) ∧ SameMem(ms1,ms2)]
                    ⇒ MemCopy(os1,ms2)
    [MemCopy(os1,ms1) ∧ Reset(os2,ms1,os3)]
                    ⇒ Restored(os1,os3)
  reward: invariant:                          // note 6
    [(∀m ∈ Memento.players):
                    MemCopy(m.orS, m.αs)]
  pattern instantiation:                      // note 7
    ⟨Originator:player; Restored, MemCopy,
                    SameMem, StateInfo, Reset⟩
  enrolling as Originator: ⟨false⟩
  enrolling as Memento:
  ⟨self = result((Originator.player).CreateMemento());
    self.orS = (Originator.player).αs⟩
  enrolling as Caretaker: ⟨true⟩   }
```

Figure 2. Memento Specification (part 1)

Notes:

1. There are three roles: Originator, Memento, and Caretaker. The '*' at the end of Memento says that any number of objects may play this role in a given instance of this pattern; Caretaker is similar; by contrast, only one object plays the Originator role.

2. In general, the state consists of components corresponding to each role. Here, only the Memento role has state, consisting of the variable orS of type OrigState, which represents the abstract type of the originator state. orS will, as specified below in the condition for enrolling as a memento, contain the state of the originator at the time this memento was created; and, as will be ensured by the specification in Fig. 4 of Memento's methods, will maintain that value[2]. Each memento object will have its own orS; this is needed since multiple objects may enroll in this role, each being a snapshot of the originator at a different time. orS is used in the reward clause below to state that each memento remains faithful, so to speak, to the state the originator was in at the time of the memento's creation.

3. We use a number of auxiliary concepts. The first, MemCopy(os, ms), says that ms is a 'memento version of', or is 'faithful to', the originator state os. One possibility, the one implied by Fig. 1, is to require ms to be identical to os. But MemCopy() allows less faithful copies such as *jpeg* representations. In any case though, MemCopy() will be a relation involving the complete states of the objects playing these roles; we use to αs notation to refer to these "application-level" states.

Next consider SameMem(). The standard description suggests that once a memento is created, no changes in its state may be made until it is used to restore the originator (or is discarded). But there is no need for such a rigid restriction. Changes in the memento state are fine so long as the memento retains 'essential' information about the originator. What 'essential' means will be specific to the pattern instance and is captured by SameMem(). SameMem() is required to be reflexive and transitive. Reflexivity, because if no changes are made, the memento continues to contain the needed information. Transitivity allows for multiple changes if each preserves the information needed. Note that no conditions are imposed on MemCopy(). The StateInfo() relation captures the relation between the memento state and the result returned by the GetState() operation of Memento; this relation will be used in reasoning about the SetMemento() operation of Originator, that being the only method that will invoke Memento.GetState().

Next, suppose at some point the originator state is os1, the memento state is ms, we restore the originator, and the resulting state is os2. Reset(os1, ms, os2) is the relation that must hold between os1, ms, and os2. The UML diagram suggests that ms and os2 are identical and os1 plays no role in this. But in general, part of the information may come from memento and part from the current originator state. Reset() allows such possibilities.

4. Next, suppose os1 is the originator state when a memento is created, the memento is later used to restore the originator, and the resulting state is os2. Restored(os1, os2) is the relation that must be satisfied by these two states. Again the simple case would be to require os1 and os2 to be identical; Restored() allows more general cases.

5. While these concepts will be specific to the pattern instance, they must satisfy two constraints. The first ensures, given the transitivity of SameMem() and the specification of the Memento role (Fig. 4), that a memento will remain in the MemCopy() relation with the state the originator was in at the time of the memento's creation. The second constraint captures the essence of the pattern: It states that

[2] orS will typically be an auxiliary variable [11] that does not appear explicitly in the actual state of the memento object(s).

if a memento was created when the originator state was os1, the originator evolves to the state os2, we reset it using the memento, and the resulting originator state is os3, *then* the states os1 and os3 must satisfy the Restored() relation. In other words, the originator will indeed have been 'restored' in a sense that is appropriate for this pattern instance (as specified by the Restored() relation).

6. Next we have the *reward* for using this pattern. Memento.players is the set of all objects enrolled in the Memento role. The reward is the invariant that the state (i.e., m.αs) of each memento will be a 'MemCopy' of the originator state at the time the memento was created.

7. Next we have the conditions that must be satisfied when an instance of the pattern is created and when objects enroll in various roles. At pattern instantiation, we are required to specify the object that will play the Originator role, and provide definitions for each auxiliary concept. This means different mementos for this instance of the pattern will all have to use the *same* auxiliary concepts; an alternative would be to require some concepts to be defined only when an object enrolls as a memento; in that case, we could have memento-specific definitions for some of the concepts.

Next is the condition that must be satisfied for another object to enroll in the role of Originator; the false condition ensures that another object cannot so enroll. The condition for an object to enroll as a memento is that the object must be the one returned by invoking CreateMemento() on the originator, and the orS component of the new memento must be set equal to the current state of the originator. The final clause states that there are no conditions on an object enrolling as a caretaker. It might seem that we could require this object to be the one that invoked the CreateMemento() operation. But that object could then pass (a reference to) the memento just created to another object which could then pass it to yet another object, etc., and each of them could act as a caretaker; thus, in effect, and as specified, any object can play this role.

Next we turn to the the Originator role in Fig. 3.

8. The initial condition when an object enrolls in this role and the invariant for the role are just true. When we consider rewriting the specification to disallow defunct mementos, we will see a more interesting initCondition.

9. Next we have specifications for the individual methods. CreateMemento() is used to create a memento. The preserves clause asserts that it does not change the state of the originator. The ensures clause asserts that the result returned is a new object; that the orS component of this new object is equal to the state of the originator; and that the originator state and the state of the new object satisfy the MemCopy() relation.

10. The specification of SetMemento() asserts that it does not change the memento involved, and that it resets the

```
roleSpec Originator {                          // note 8
 initCondition: true;
 invariant: true;
 methods:
  Memento CreateMemento():                     //note 9
   requires: true;
   preserves: αs;
   ensures:
     [ (new result) ∧ (result.orS = self.αs)
       ∧ MemCopy(self.αs, result.αs) ]
  void SetMemento(Memento m):                  //note 10
   requires: [m ∈ Memento.players];
   preserves: m.αs;
   ensures: [Reset(αs@pre, m.αs, αs)]
  others:                                      //note 11
   requires: true
   ensures: true   }
```

Figure 3. Memento Specification (part 2)

originator state so that the originator state at the start of the operation, the state of the memento, and the final state of the originator satisfy the Reset() relation[3].

11. The class of the object playing this role may have additional methods. In general, we need to ensure that these other methods behave in ways that do not violate the intent of the pattern. For this role, as specified by the "others" part of the specification, there are no non-trivial conditions; but, as we will see, for the Memento role, we have an interesting requirement.

Next consider the Memento and Caretaker roles, Fig. 4.

12. The initCondition and invariant for this role are, as in the case of the Originator role, just true.

13. GetState() may be used only by the originator, as specified by the requires clause. This method returns appropriate information needed to restore the originator state; what 'appropriate information' means is defined by StateInfo(), and the ensures clause requires that the result returned by this method and the memento state satisfy this relation. This will be used in reasoning about SetMemento() of the object enrolling as the originator, to ensure that it does meet the specification in Fig. 3.

The ensures clause of other methods, as well as that of GetState(), require that the state when they finish and the state at the start satisfy the SameMem() relation; this, in conjunction with the first constraint specified in Fig. 2, ensures that the role's invariant, hence the reward clause of Fig. 2, will be satisfied.

The standard UML diagram of the pattern also includes a method SetState() for this role. This is intended to be used by CreateMemento() of Originator when

[3]"@pre" is the OCL [17] notation for referring in the post-condition, to the values of variables at the start of the method execution.

```
roleSpec Memento {
 initCondition: true;                              // note 12
 invariant: MemCopy(orS, $\alpha s$);
 methods:
  GetState():                                      //note 13
   requires: (caller = Originator.player);
   preserves: orS;
   ensures: StateInfo(result, $\alpha s$)
            $\wedge$ SameMem($\alpha s$@pre, $\alpha s$);
  others:
   requires: true
   preserves: orS;
   ensures: SameMem($\alpha s$@pre, $\alpha s$);    }
roleSpec Caretaker { }                              // note 14
```

Figure 4. Memento Specification (part 3)

creating the memento. But as long as CreateMemento() works as specified in Fig. 3, there is no reason (from the point of view of the pattern) to impose conditions on *how* it works. Hence we assume the SetState() method is not part of this role (and the corresponding class).

14. The specification of Caretaker is empty since the pattern does not require any particular methods of this role, nor are there any conditions on its "other" methods.

Next we briefly consider how to account for *defunct* mementos. Introduce a variable, liveMem, in the state of the Originator (currently null, Fig. 2) to keep track of the mementos that are still live:

Originator: Seq[Memento] liveMem;

The initCondition of Originator will become (liveMem $= \langle\rangle$), since no mementos have yet been created. Next, we add to the ensures of CreateMemento() (Fig. 3), the following clause:

(liveMem = liveMem@pre $\triangleright$ result)

This ensures that when a memento is created, it is appended ("$\triangleright$" denotes append) to the end of liveMem. Next, the requires of SetMemento() will be changed to (m $\in$ liveMem), to check that the memento being used to restore the originator is live. Finally, the ensures of the same operation will have the following clause added:

(liveMem = Prefix(liveMem@pre, m))

Prefix() takes a sequence and an element of the sequence as its arguments, and returns a sequence consisting of the elements that precede the given element. This ensures that the memento being used to perform the restoration, as well as all mementos that follow it in liveMem, are removed from liveMem since they are now defunct.

4 Related Work

One approach to making the descriptions of patterns more precise has been to develop extensions to UML. Vlissides [16] describes an extension based on Venn diagrams, which identifies the patterns involved in a design, as well as the classes used to implement each pattern. Vlissides also describes *role annotations* to identify the roles associated with each class. Dong [3] extends this to annotate methods and attributes to identify their role in each pattern. Reenskaug [12] uses the notion of *role models* to describe how the roles of a pattern interact with each other. But these authors do not address the question of formally characterizing the behavioral aspects of the pattern.

In Eden *et al.*'s [5, 6] approach, patterns are meta-level programs operating on an intermediate system representation. Aspects that vary among instances of the pattern are deferred to the designer. Yau *et al.* [18] describe a representation that identifies the participants, their structural properties (such as method signatures), and interactions. Individual classes are bound to specific participant descriptions, structural conformance is automatically verified, and wrappers generated to provide the prescribed interactions. But neither approach specifies the behavioral responsibilities that must be satisfied when applying the pattern.

Mikkonen [10]'s work focuses on behavioral concerns. He formalizes design patterns by specifying the classes involved, including their abstract models and the relations. Some behavioral aspects are specified using DisCo, an action system notation. Although the approach allows behaviors to be specified precisely, *how* the behaviors are to be achieved is not part of the specifications in [10]. For example, the specification of Memento in this notation can be satisfied by the originator saving its own state, rather than creating a separate memento object; but since the main point of the pattern is the use of the memento to save and later restore the originator, such a specification does not seem to capture the essence of the pattern. Another problem with the approach is that it is somewhat inflexible. Thus the Memento spec would require the entire state of originator to be saved; no changes would be allowed in the state of memento once it is created (assuming that we use a separate memento object, rather than having the originator save the state internally); etc.

5 Discussion

Our goal was to show how design patterns may be precisely and unambiguously specified by using Memento as a detailed case-study. Specifications of the type we propose can complement standard informal descriptions and help ensure that all designers have a common understanding of the requirements that must be met by various parts of their system if the pattern is to be applied faithfully, as well as the behaviors that the system will exhibit as a result.

The specification of a pattern in our approach consists of conditions that must be satisfied when the pattern is instantiated, and when an object enrolls to play a role; *requires* and *ensures* clauses for the methods of each role, including other methods not required by this pattern but that may be included in the class of the object; an *invariant* for each role; and a pattern-level *invariant* relating the different roles. These assertions are specified in terms of state components associated with the various roles, and in terms of *auxiliary* concepts and relations. The essence of the pattern is captured by the invariant relating the different roles. In the case of the Memento pattern, this invariant (given the various constraints and the specifications of the individual roles), specified that the state of each memento remains 'faithful' to, i.e., satisfies the MemCopy() relation with, the state the originator was in at the time the particular memento was created. Similarly, for the Observer pattern [14], the invariant specifies that the state of each observer remains consistent with the state of the subject, as the latter's state evolves during execution.

An essential aspect of our approach is that it preserves, and indeed, somewhat surprisingly, helps identify ways to *extend*, the flexibility of the pattern. Consider the *graphics editor* of [8], in which the originator state, consisting of the nodes and edges of the graph, is saved in a memento for later restoration. One possible notion of 'restoration', the one used in [8], would require each node and each edge to be at exactly the *same positions* they occupied originally; we can capture this notion in our approach by defining the concept of Restored(os1, os2) appropriately. On the other hand, if restoration only required the nodes to be restored to their original locations and the edges reset to ensure the *same connectivity* as before but not necessarily occupy their original locations, the designer can do so by appropriately loosening the definition of Restored(). Indeed, an even weaker notion might only require that all the nodes are restored to their original locations, with no conditions on the edges; this too is handled by redefining Restored() appropriately. In a sense, the Restored() concept *encourages* the discovery of such new notions of restoration, making the pattern more flexible than ever. Of course, once we discover these additional dimensions of flexibility, we can update the informal descriptions appropriately; but the fact remains that it was the work of developing the specifications that led to the identification of these dimensions.

We conclude with a couple of pointers to future work. We plan to develop the specifications of a number of other patterns. Patterns for concurrent/distributed systems would be of special interest, raising issues not present in sequential patterns. We also intend to work on ways to verify that a given system built using a pattern meets the requirements of the pattern's specification. One problem is that, in a sense, the use of the pattern is not explicit in the system code; rather, it is in the 'eye of the designer'. A way around this would be to require the system to be *annotated* suitably to provide information about the pattern instantiations, the definitions of the auxiliary concepts, etc. Given such an annotation, we can then try to show the system does meet the requirements of the pattern specification.

References

[1] K. Beck and R. Johnson. Patterns generate architectures. In *Proceedings of the Eighth ECOOP*, pages 139–149, 1994.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A system of patterns*. Wiley, 1996.

[3] J. Dong. UML extenstions for design pattern compositions. In C. Mingins, editor, *Proc. of TOOLS, in Special Issue of Journal of Object Technology, vol. 1, no. 3*, pages 149–161, 2002.

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

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

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

[7] T. Elrad, R. Filman, and A. Bader, editors. *CACM Special Issue on Aspect Oriented Programming*. ACM, Oct. 2001.

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

[9] R. Johnson. Components, frameworks, patterns. In *ACM SIGSOFT Symposium on Software Reusability*, pages 10–17, 1997.

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

[11] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(1):319–340, 1976.

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

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

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

[15] T. Taibi and D. Ngo. Formal specification of design patterns – a balanced approach. *Journal of Object Technology*, 2(4):127–140, July–August 2003.

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

[17] J. Warmer and A. Kleppe. *The Object Constraint Langauge*. Addison-Wesley, 1999.

[18] S. Yau and N. Dong. Integration in component-based software development using design patterns. In *The Twenty-Fourth Annual International Computer Software Applications Conference*, pages 369–, Taipei, Taiwan, October 2000.