From Architectural Patterns to Architecture-Based Development

Neelam Soundarajan
Computer Sc. & Eng. Dept.
Ohio State University
Columbus, OH 43210, USA

Jason O. Hallstrom
Computer Science Dept.
Clemson University
Clemson, SC 29634, USA

8 September 2005

Abstract

Architectural patterns describe reusable architectural fragments that have proven valuable in a range of system contexts. While their impact on software practice has been significant, their application has been limited to the early stages of the software lifecycle. In this paper, we present an approach to reinforcing the productivity and reliability benefits provided by architectural patterns *across* the lifecycle. The approach is based on a catalog of precise pattern specifications, and is supported by runtime monitoring and architecture exploration tools. We describe the basic elements of the specification approach, the supporting software tools, and the benefits that they provide. We illustrate the approach with a standard architectural pattern.

1 Introduction

Software practitioners are aware of the benefits provided by *architecture-centric* development processes [2, 5, 9]. Bertolino *et al.* [3], for example, describe how to construct test cases for validating the conformance of a system to its SA based on a specification of the architecture's component dynamics. Moreira *et al.* [14] consider how SA specifications can provide guidance in modifying a system to accommodate evolving requirements. Stafford *et al.* [10] present dependence analysis tools for automating the discovery of anomalies in a system based on its SA specification.

In this paper, rather than considering a system's architecture as a whole, we outline a development process based on architectural *patterns* [4, 6, 15] — reusable architectural fragments codified in a standard format. The approach is based on a catalog of formal pattern specifications that complement existing informal descriptions of the patterns. As we will see, these specifications support an architecture-centric development process that provides productivity and reliability benefits across the lifecycle. Tools based on the approach further enhance these benefits.

In recent years, *architecture description languages* (ADLs) have gained increased acceptance. Garlan provides a survey of several such languages in [7]. While some have considered the use of such languages in specifying patterns, our approach is unique in several respects. First, in these approaches, the specifications are typically developed in the context of particular pattern applications. By contrast, in our approach, the specifications apply to any systems in which the corresponding patterns might be used. Second, system architects can tailor the specifications as appropriate by tuning the relevant parameters. As in some ADLs, some of these parameters express basic name mappings. Others, however, capture essential structural and behavioral aspects of the system, ensuring that the specifications are applicable to a wide range of systems. Third, our approach is supported by automation tools that preserve the architectural integrity of a system. As we will see, these tools, combined with the appropriate pattern specifications and parameter definitions, provide software architects and engineers with guidance on how a system may be modified or extended without

violating its architectural integrity. In this paper, we present an overview of the specification approach, the supporting tool suite, the process model enabled by these elements, and the benefits that should be expected as a result.

2 Contracts for Architectural Patterns

Design-by-contract [13] is a fundamental concept in software engineering. The essence of the approach is to have a precise specification (or "contract") for each class. For each method of the class, the specification characterizes the conditions that must be satisfied by callers of the method (the "pre-conditions"), and the conditions that the method will satisfy in return (the "post-conditions"). The approach affords designers with the flexibility to modify (or completely replace) their class implementations, as long as they respect the appropriate contract requirements. Further, it allows designers to develop their systems without concern for the constituent component implementations; only the contracts are necessary. Adapting a similar approach to architectural patterns offers a number of advantages and can indeed form the basis of a powerful software development process.

At the same time, however, there are important challenges to be overcome. Much of the power of patterns arises from our ability to customize them as appropriate to a wide range of systems. Our contracts, which must be precise, must not compromise this flexibility. To illustrate the challenges, consider the classic *Model-View-Controller* (MVC) pattern, originally described by Krasner and Pope[1] [11]. The pattern is used primarily in the design of interactive applications with flexible user interfaces. Common issues encountered in such designs include the need for multiple interfaces that cater to different needs, "look-and-feel" variations across platforms, etc. These issues are difficult to address if the user interface code is tightly coupled to the rest of the system.

Hence the MVC pattern divides the application into three *roles*: Model, View, and Controller. During execution, various objects *play* these roles. The model[2] is responsible for implementing the functional core of the system. Each view obtains data from the model, and presents the information to the user in an appropriate format. Multiple views of a given model may be supported, each with an associated controller. The controller associated with a view accepts input from the user, and performs the appropriate action on either the model or the view. All interactions between the user and the system are mediated by the relevant controllers. If the user modifies the model through one of the controllers, *all* of the model's views must be updated.

To achieve this, the model maintains a set of references to the relevant views and controllers. These objects are collectively referred to as its observers. observer objects are added to the set by invoking the model's attach() method. Similarly, observers are removed by invoking the detach() method. When the state of the model changes, its Notify() method is invoked, which in turn invokes Update() on each attached observer. When an observer's Update() method is invoked, it retrieves the current data from the model and updates itself appropriately.

The standard UML diagram for the pattern, shown in Fig. 1, and the informal description above leave some important questions unanswered. The description in POSA-1, for example, notes that "changes to the state of the Model trigger the change-propagation mechanism". But which changes? Any change in any bit of the model? Surely not. Suppose that a new view invokes Model.Attach(). The state of the model will certainly be modified; the view will be added to its set of attached observers. Do the other observers need to

[1] Our discussion is also based, in part, on the one in the "POSA-1" book by Buschmann *et al.* [4].

[2] We use names starting with uppercase letters for roles, and the corresponding lowercase names to refer to the objects that play these roles.

be updated because of this change? Probably not. The other objects are interested in *significant* changes in the state of the model. But how do we determine those changes that are significant?

Consider another example. The informal description states that "[w]hen the Update() procedure is called, a view retrieves the current data values from the model and puts them on the screen". Which data values? And what does it mean to "put [the data values] on the screen"? What if the view does not use a screen?

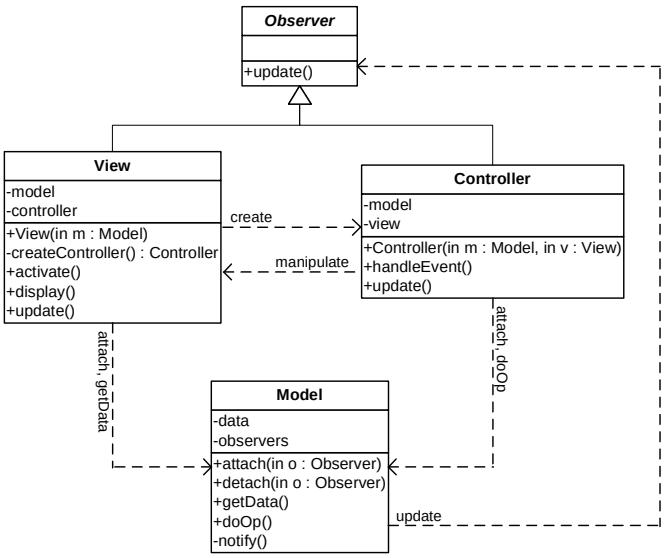

Figure 1: The MVC Pattern

In other words, what are the requirements that each of the roles must satisfy to ensure that the MVC pattern is used as intended? If we do not have a precise answer to this question, different designers may come to different, mutually inconsistent conclusions. A designer might, for example, assume that a particular change in the model is insignificant, and omit the call to Notify(). A different designer, one responsible for a particular view implementation, might expect notifications when such a change occurs. The problem may become more visible over time as the system evolves. Initially, the view might ignore the portion of the model under modification. Hence, the missing call to Notify() won't be noticeable. At a later time, the view might be enhanced to display additional information, including the previously ignored model state. Now, the bug in the Model class will suddenly manifest itself as a defect in the View!

Our goal is to provide precise answers to the kinds of questions raised above. Given such answers, the risk that individual designers will interpret their implementation responsibilities in mutually inconsistent ways will be minimized. Moreover, given precise pattern contracts, it is possible to develop tools that can monitor a system's runtime behavior to determine whether the relevant requirements are satisfied. Such tools are especially valuable during system maintenance and evolution and help to preserve architectural integrity.

Pattern contracts also make it possible to construct architecture *exploration* tools. These tools are important given that many systems have transparent architectures; they are not explicit in the source code. Instead, the architectures manifest themselves in the runtime interactions of their constituent objects. In the case of a system built using MVC, for example, there are no syntactic entities labelled "MVC". The pattern becomes evident only when we focus on the interactions among the objects playing the various roles. We will return to the possibility of architecture exploration tools later in the paper.

There is, however, a risk in providing precise answers to the types of questions raised by informal pattern descriptions. If, for example, we provide one specific answer to the question of what constitutes a significant change in a model, the pattern may not be applicable in systems that take a different perspective. The same is true if we provide one specific answer to the question of what information should be displayed by each view. Indeed, even in the same system, it is possible to have two different view objects that have mutually incompatible notions of what should be displayed. In this case, it seems that no matter how we write the contract for MVC, these two View classes cannot both simultaneously respect the contract. In other words,

we seem to be faced with a fundamental conflict between precision and flexibility.

3 Having Your Cake and Eating It Too!

In general, there are two ways of resolving the conflict between precision and flexibility. First, we could write the contract in such a way that it explicitly accounts for all possible ways of satisfying it. For example, the contract could specify all possible ways that a view might display the current state of the model. Clearly, this is impractical. The second way of resolving the conflict, the one that we adopt, is to *parameterize* the contract appropriately. The contract for a given pattern will be written in terms of various *specialization* parameters. The *values* of the parameters will not be defined in the contract, since they will vary from one system to another. Instead, the values appropriate to a given system will be defined in a *subcontract* corresponding to that system's use of the pattern. When parameter values are "plugged-into" the pattern contract, the result will be a specialized contract that describes the architecture of the system in question. (Or rather, that portion of its architecture related to the pattern.) Thus, in effect, the pattern contract describes the pattern as it applies to all systems built using it. A subcontract tells us how the pattern is specialized for use in a particular system.

The parameters used in a pattern contract must allow for two distinct types of variation. The first is with respect to the *structure* of the system. For example, it may be that in an MVC-based system, a class playing the Model role stores references to its observers in a *linked list* rather than in a *set* (as indicated by the UML diagram of the pattern). Or it may be that the method corresponding to Model.Attach() is instead named, say, Observe(). Name-based variation is similar to that allowed in ADLs such as Darwin [12] and Wright [1]. But we need to allow for more general variation, such as allowing a linked list to be used in place of a set. We do this by requiring that each subcontract provide suitable pieces of code that map the state of the class playing each role to the corresponding state elements defined by the role. Thus, in the case of the class playing the Model role in the previous scenario, the relevant code in the subcontract would return a set containing the elements stored in its linked list.

The second type of parameter is rather novel, and allows for *behavioral* variation. In the case of MVC, for example, as we noted earlier, the notion of what constitutes a *significant change* in the state of a model object varies from one system to another. To allow for this, we introduce the notion of an *auxiliary concept*. An auxiliary concept is a relation involving the states of one or more objects playing roles in a given pattern. The contract for MVC, for example, defines the auxiliary concept *Modified*$(m1, m2)$, where $m1$ and $m2$ are two states of a model object. *Modified*$(m1, m2)$ represents the notion of a *significant change* in the model. If the state of the model were to change from $m1$ to $m2$, and *Modified*$(m1, m2)$ were to evaluate to *true*, the Notify() method must be invoked to update the attached observers. This requirement will be part of the MVC contract, but the definition of *Modified*() will be provided in the corresponding subcontract as appropriate to the system in question.

The MVC contract also declares *Consistent*$(m1, v1)$, where $m1$ represents the state of a model object and $v1$ represents the state of a view object. Rather than requiring that the Update() operation "put the data values [of the model] on the screen", the contract requires that the execution of Update() bring the view into a state that is *consistent* with the model. But again, the definition of *Consistent*() will vary from system to system, and will therefore be deferred to a subcontract. Hence, auxiliary concepts allow us to specify, as part of the pattern contract, precise requirements that apply to all systems, while allowing the behaviors to be tailored to the needs of individual system.

Fig. 2 provides a pictorial representation of our approach. The box in the upper left, labelled architectural pattern, represents a pattern P, as described informally. The roles *R1*, *R2* of P are represented by the dashed circles. The dotted structure enclosing the two roles represents the informal description of how the roles are expected to interact. The box in the lower left, labelled pattern contract, represents the contract for P, and will be part of the *pattern contract catalog* shared by the community. The role contracts for the individual roles specify the conditions that the methods of the individual roles must meet. The most interesting aspect of most patterns, however, is not the behaviors of the individual roles, but the *interactions* among them. The *interaction contract* defined by the patttern contract specifies these interactions and the resulting behavior that they ensure for the system. Typically, the interaction contract will involve the states of all the objects playing roles in the pattern, and will be expressed in terms of auxiliary concepts. We will see an example in the next section.

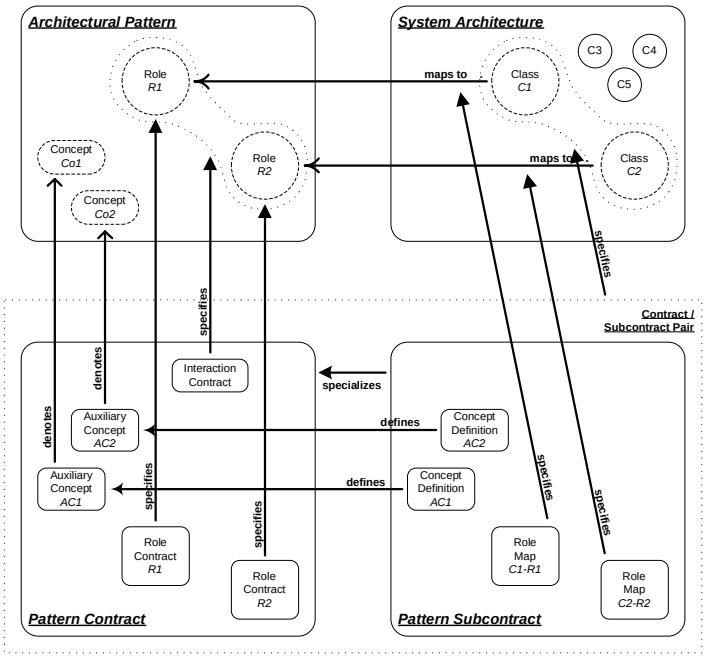

Figure 2: Pattern Contracts and Subcontracts

The two boxes on the right side of Fig. 2 represent an application of P in a particular architecture, and the corresponding subcontract. The box on the top right shows the constituent classes of the architecture.The arrows going to the box on the top left indicate the assignments between classes and roles. The box in the bottom right represents the subcontract corresponding to this specialization of the pattern. The *role maps* specify how the fields and methods of each class map to the fields and methods of their respective roles. In addition, auxiliary concept definitions have to be provided for each of the concepts that are declared in P's contract. Note that for a different system built using P, the left side of this figure will remain the same.

There are two important properties that are not shown in Fig. 2. First, although the auxiliary concepts will be defined in a subcontract, these definitions cannot be arbitrary. Preserving the intent of the pattern across architectures requires that auxiliary concept definitions satisfy certain constraints. These constraints will be specified as part of the pattern contract. Second, the role contracts will have to impose requirements not only on the *named* methods required by the pattern (*e.g.*, `Attach()`, `Notify()`), but also on those methods provided by classes playing the roles that do not correspond to methods required by the pattern. We refer to these methods as "other" methods or "unnamed" methods. In general, if these other methods behave in arbitrary ways, the intent of the pattern may be violated even if the named methods behave as specified in the pattern contract. To prevent such problems, each role contract includes a specification for the *other* methods of its role. These points will be illustrated in the next section.

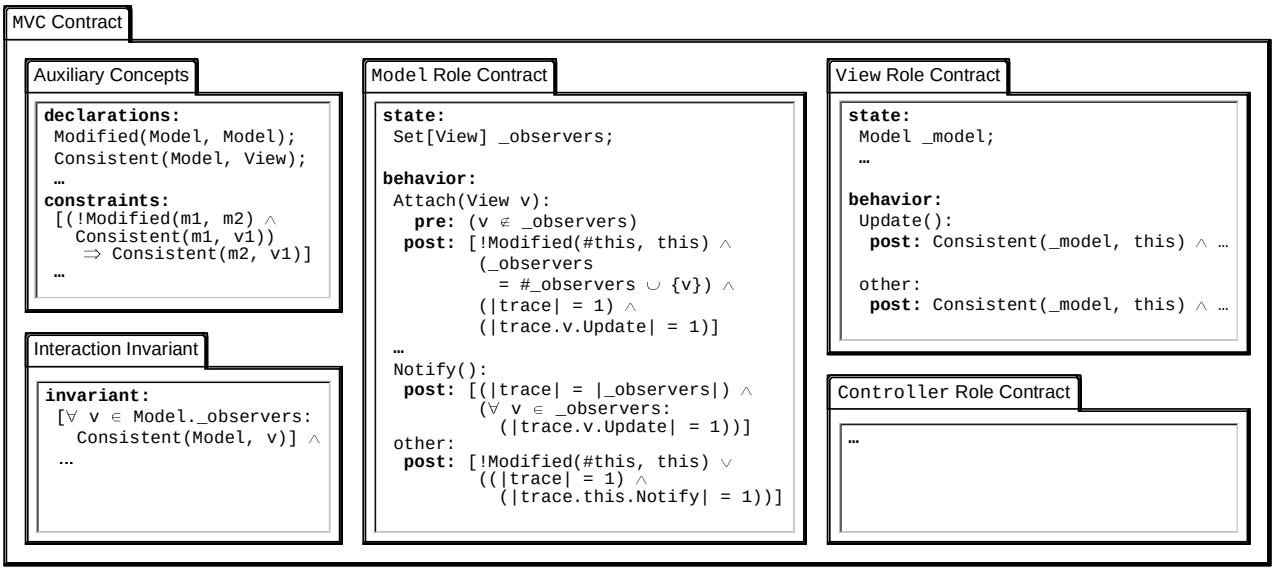

Figure 3: MVC Contract

4 Case Study

In this section, we consider key portions of the MVC contract, shown in Fig. 3, and a simple system built using the contract. We focus on the Model and View roles, the relevant auxiliary concepts, and the constraints imposed on those concepts.

Each role contract specifies pre- and post-conditions for the role's *named* and *unnamed* methods. These specifications are expressed in terms of the relevant method parameters and the fields defined by the role. In addition, a post-condition may impose requirements on the trace of a method. The trace of a method is a record of the calls it makes during its execution. In the case of Model.Notify(), MVC requires that it invoke Update() on each attached observer. This is what the post-condition states. The number of calls recorded in the trace must equal the number of elements in _observers. And for each view to which there is a reference v in _observers, there must be exactly one call to its Update() method. The *other* specification states that unnamed methods must either not *Modify*() the model, or must invoke Notify(). Note the interplay between flexibility and precision. The requirement that *significant modifications* trigger a notification is captured precisely. Deferring the definition of this concept to a subcontract ensures flexibility.

The specification of Update() in the View role contract states that Update() must bring the view into a state that is *Consistent*() with the state of the model. Unnamed methods may alter the state of the view, but must preserve *consistency*. Note again the interplay between flexibility and precision. Unnamed methods are required to preserve *consistency*, ensuring precision. The definition of consistency is allowed to vary from system to system, ensuring flexibility.

Assuming that the role contract requirements are satisfied, what can a designer conclude about a group of objects interacting according to the pattern? when control is outside of the participating objects, the state of each view *observing* a model will be *consistent* with that model. This is specified by the *interaction invariant*.

Finally, consider the constraint imposed on definitions of these concepts. Suppose that a model is in

state m1, and one of its views is in state v1, where *Consistent*(m1,v1) is satisfied. Suppose that the state of the model changes to m2, where *Modified*(m1,m2) is *false*. As a result, according to the *other* specification in the Model role contract, the Notify() method might not be invoked. Hence, the state of the view object will remain unchanged. Suppose that *Consistent*(m2,v1) is *false*. Now we have a problem! Although the interactions required by the MVC pattern have been satisfied, the interaction invariant is no longer satisfied! The problem is not in how the objects are interacting, but rather in the incompatible definitions of the two concepts. The constraint specified forbids such incompatibilities and it must be respected by all subcontracts.

Consider a simple example involving an Elevator class whose instances represent an elevator. Its state includes _np, the number of people in it, and _nf, the number of the floor it is currently on. We have a Display class whose instances display the current floor number of the elevator. Any number of displays may be attached to an elevator. A reasonable implementation of Elevator would invoke its Notify() method whenever the elevator goes to a new floor since the value of _nf changes.

Suppose during system evolution, a SafetyDisplay class is created. Instances of this class will display both the floor number and the number of people in the elevator. When this class is implemented and tested, everything seems to work well. But reports start coming in that SafetyDisplay is *occasionally* displaying incorrect information about the number of people.

The problem is that Notify() is invoked only when the value of _nf changes. This is fine for display objects. But safetyDisplay objects are also concerned with the value of _np. And this part of the elevator state can change even if _nf has not changed — perhaps because the DoorOpen() method admits people into the elevator. This was overlooked during testing of SafetyDisplay because, by coincidence, whenever it was tested, the value of _nf had also changed.

With our approach, the error would have been caught earlier. When the designers defined *Modified*() in the MVC subcontract, they would have considered a change in _np to be significant. After all, if it was not, why include it in Elevator? Therefore, the implementation of DoorOpen() would be seen as obviously incorrect since it does not meet the requirements of Model.other. (The state may be *Modified*() without a call to Notify().) A team that uses formal reasoning would have identified this and revised the Elevator class appropriately. Or, using the *MonGen* tool described in the next section, testers of the original implementation would likely have caught the error.

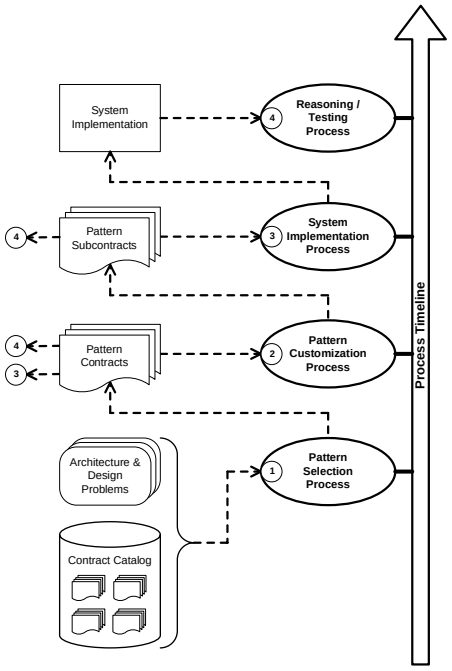

Figure 4: Pattern-Based Software Lifecycle

5 Software Lifecycle

Let us now consider how our approach supports the software lifecycle, as shown in Fig. 4. In the first step, architects, domain experts, and designers decide on the main architectural features of the system, and are guided by known patterns from the pattern catalog. This reflects current practice, with the important addition of precise pattern specifications. As a result, the pattern selection process is better informed, and the resulting architectural constraints are unambiguous. In the next step, designers and implementers determine

the details of the system, including the mappings of classes to roles. These details serve as the basis for developing the corresponding subcontracts.

The contracts and subcontracts serve two distinct purposes. First, they provide documentation that architects and designers can reference in understanding the overall architecture of the system, and the high-level details of the architecture's realization. Second, they provide clear requirements that the system implementers must meet; this is shown in step (3). This ensures that implementers have a clear set of behavioral requirements to guide them in implementing the appropriate classes.

During the testing phase (step (4)), the contracts and subcontracts characterize the assertions that must be tested. We have implemented a tool, *MonGen*, that generates appropriate monitoring code in *AspectJ* based on the contracts and subcontracts underlying a given architecture. When the aspect code is woven with the original system code, it checks, at appropriate points, that the assertions specified in the contracts are satisfied. If the appropriate assertions are satisfied, execution continues normally; otherwise, appropriate warning messages are displayed, alerting designers of the architecture violation.

Space limitations preclude a detailed discussion of *MonGen*, but one point is worth noting. *AspectJ* primitives allow us to *intercept* program execution when specified methods are invoked. This enables the tool to save information about the *traces* associated with various methods. This information is essential for checking such assertions as the post-condition of `Model.Notify()` in the MVC pattern. Full details concerning *MonGen*, as well as instructions for using the tool, are available at our website [8].

MonGen is only one element of the tool suite that we are constructing. We are currently working on an extension to *MonGen* that records additional pattern-related information. This information will be used to provide *pattern-centric* visualizations of a system's runtime behavior. A user might, for example, focus only on the interactions of objects participating in particular roles of a particular pattern. We expect that this tool will be of great value during system maintenance and evolution. It will help those involved with this phase of the lifecycle to discover, in a natural way, the intended system architecture, and will help them remain faithful to that architecture as the system evolves.

Author Biographies

Neelam Soundarajan is an Associate Professor in the Department of Computer Science and Engineering at Ohio State University. **Jason O. Hallstrom** is an Assistant Professor in the Department of Computer Science at Clemson University. Both authors focus on tools and techniques for improving system reliability.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans on Softw Eng and Meth*, 6(3):213–249, 1997.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2004.

[3] A. Bertolino, P. Inverardi, and H. Muccini. Formal methods in testing software architectures. In *Formal methods for software architectures*, LNCS 2804, pages 122–147. Springer, 2003.

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

[5] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

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

[7] D. Garlan. Formal modeling and analysis of software architecture: ”c”omponents, connectors, and events. In *Formal methods for software architectures*, LNCS 2804, pages 1–24. Springer, 2003.

[8] J Hallstrom. Patterns in the software lifecycle. http://dsrg.cs.clemson.edu/.

[9] C. Hoffmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, 2005.

[10] J.Stafford, A. Wolf, and M. Caporuscio. Application of dependence analysis to software architecture descriptions. In *Formal methods for software architectures*, LNCS 2804, pages 52–62. Springer, 2003.

[11] G. Krasner and S. Pope. A cookbook for using the mvc interface paradigm in smalltalk-80. *J. of Object Oriented Programming*, 1(3):26–49, 1988.

[12] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proc. of 5th Eur. Softw Eng Conf*, pages 137–153. Springer, 1995.

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

[14] A. Moreira, J. Fiadeiro, and L. Andrade. Evolving requirements through coordination contracts. In *CAiSE*, LNCS 2681, pages 633–646. Springer, 2003.

[15] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-oriented software architecture: Patterns for concurrent and networked objects*. Wiley, 1996.