The Journal of Systems and Software 83 (2010) 1735–1758

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

How do architecture patterns and tactics interact? A model and annotation

Neil B. Harrison[a,b,*], Paris Avgeriou[b]

[a] Department of Computer Science, Utah Valley University, Orem, UT,USA
[b] Department of Mathematics and Computing Science, University of Groningen, Groningen, The Netherlands

ARTICLE INFO

Article history:
Received 10 June 2009
Received in revised form 20 April 2010
Accepted 20 April 2010
Available online 15 June 2010

Keywords:
Software architecture
Patterns
Quality attributes
Tactics

ABSTRACT

Software architecture designers inevitably work with both architecture patterns and tactics. Architecture patterns describe the high-level structure and behavior of software systems as the solution to multiple system requirements, whereas tactics are design decisions that improve individual quality attribute concerns. Tactics that are implemented in existing architectures can have significant impact on the architecture patterns in the system. Similarly, tactics that are selected during initial architecture design significantly impact the architecture of the system to be designed: which patterns to use, and how they must be changed to accommodate the tactics. However, little is understood about how patterns and tactics interact. In this paper, we develop a model for the interaction of patterns and tactics that enables software architects to annotate architecture diagrams with information about the tactics used and their impact on the overall structure. This model is based on our in-depth analysis of the types of interactions involved, and we show several examples of how the model can be used to annotate different kinds of architecture diagrams. We illustrate the model and annotation by showing examples taken from real systems, and describe how the annotation was used in architecture reviews. Tactics and patterns are known architectural concepts; this work provides more specific and in-depth understanding of how they interact. Its other key contribution is that it explores the larger problem of understanding the relation between strategic decisions and how they need to be tailored in light of more tactical decisions.

1. Introduction

The architecture of a software-intensive system is the foundation upon which the system is implemented to satisfy the system requirements (note that in this paper, references to ""architecture" refer to the architecture of software-intensive systems; see IEEE, 2000). The requirements include both functional and non-functional requirements. The non-functional requirements, or quality attribute requirements, are extremely important, and can be even more important than the more visible functional requirements (Barbacci et al., 2003), and even drive software architecture design (O'Brien et al., 2005). The architecture is designed by making a number of design decisions in order to influence the requirements. Two types of important design decisions are the application of architecture patterns and tactics.

Architecture patterns are common architectural structures which are well understood and documented (Buschmann et al., 1996; Harrison et al., 2007; Schmidt et al., 2000; Shaw, 1988). Each pattern describes the high-level structure and behavior of general software systems and aims to satisfy several functional and non-functional requirements. Architecture patterns contain the major components and connectors of the system to be built. Most modern software architectures use one or more architecture patterns (Harrison and Avgeriou, 2008a). (Note that patterns exist at levels other than architecture, such as design; in this paper, references to patterns refer to architecture patterns.) Architecture patterns are chosen in response to early design decisions, including decisions about how to satisfy functional requirements, non-functional requirements (quality attributes), and physical constraints (such as physical distance between user and service provider.) Thus architecture patterns provide the major structures in which multiple design decisions are realized.

An important class of design decisions is those that concern quality attributes; such decisions are often called tactics (Bass et al., 2003). A tactic is a design decision that aims to improve one specific design concern of a quality attribute. For example, a design concern concerning the quality attribute "security" is how to prevent attacks on the system; an associated design decision (tactic) is to authenticate users. As with patterns, the structures and behavior of the tactics may also shape the architecture, but usually at a smaller scale; this is natural as tactics concern a single quality attribute while patterns address a number. Tactics usually become part of a pattern alongside other structures.

* Corresponding author at: Department of Computer Science, Utah Valley University, 800 West University Parkway, Orem, UT 84058, USA.
E-mail addresses: neil.harrison@uvu.edu (N.B. Harrison), paris@cs.rug.nl (P. Avgeriou).

0164-1212/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2010.04.067

Because tactics must be realized within architecture patterns, the relationship between the two needs special study. Architecture patterns and tactics have an influence on each other as their structure and behavior need to co-exist within the architecture. This is a common case of dependencies between design decisions: former decisions constrain subsequent decisions. We need to distinguish between two cases:

- Brownfield development: in the case of evolving legacy systems the patterns are already in place and the architect may need to apply additional tactics to improve the quality attribute requirements. In this case, a tactic may require small or large changes to the patterns in order to fit in the architecture for its implementation.
- Greenfield development: in the case of designing systems from scratch, an architect can first select patterns and then fit the tactics (top-down), or first select tactics and then see which patterns can incorporate them (bottom-up) or select both patterns and tactics iteratively (meet-in-the-middle). The second case is the most common as the architect usually starts making minor design choices by selecting tactics and other structures (e.g. for functional requirements and physical constraints of the system), then selects a pattern that can incorporate these choices and subsequently selects more tactics and other structures.

In both cases we need information about how the pattern-related and tactic-related decisions influence one another. The information may be used to modify the pattern, or if one has not selected an initial pattern yet, it can be used to understand the tradeoffs of selecting one pattern versus another. Specifically we need information about how much a pattern must be changed in order to implement certain tactics. Of course, this also reflects how easy or difficult it is to implement the tactics in the pattern.

In practice, for any given tactic, we may not know which parts of the pattern make it easy or difficult to implement that tactic. Furthermore, architects and developers often do not understand why a tactic is easy or difficult to implement in a given pattern. This makes it difficult to make informed choices about alternative tactics, and to assess the costs and benefits of tactics to achieve quality attributes. In short, their ability to effectively satisfy the critical quality attributes for a system is compromised.

Several sources (Cloutier, 2006; Harrison and Avgeriou, 2007a) have documented that certain architecture patterns have benefits or liabilities for certain quality attributes. However, such descriptions are generally very high level; while they give general guidance, they do not explain the positive or negative consequences in sufficient detail to identify specific parts of the pattern that are impacted by certain quality attribute tactics, and exactly what that impact consists of. In fact, current pattern documents do not even mention tactics at all.

We have studied one quality attribute (reliability) in depth to understand the interaction of its tactics with the most well-known architecture patterns (Harrison and Avgeriou, 2008b). We found that there are many interactions, and that there are several types of interactions, varying from pattern to pattern. The interactions between patterns and tactics are indeed rich and varied.

Therefore, without detailed knowledge of the architecture patterns used and how they interact with the quality attribute tactics, an architect can easily miss important interactions. Furthermore, without an understanding of the types of interactions, an architect will not even know what to look for. In addition, one will not easily understand how and why a pattern is modified in order to satisfy a quality attribute.

In order to help architects and designers understand the intricate relations between a system's architecture design decisions and its quality attributes, we describe a model of interaction between architecture patterns and tactics. This paper describes and categorizes these interactions, shows how they apply in architectural design, and provides a way to annotate architectural diagrams so that others can easily understand where these interactions take place.

By applying this model, an architect can learn the nature of the changes to a software architecture that must be made in order to implement desired quality attributes. This can lead to:

- Better ability to assess whether a particular architecture pattern is appropriate for the system being designed.
- Better ability to estimate the development effort required – instead of guessing at how a quality attribute will be implemented, one can see the architectural components that must be changed or added to implement it. That means less guesswork and fewer surprises later in development.
- Ability to evaluate alternative approaches to implementing quality attributes, namely evaluating alternate tactics that accomplish the same thing. You might have considered one, but may pick a different one because it is easier to implement in the architecture.
- Reuse of this knowledge through the use of known patterns and tactics.
- Better understanding of the architecture; namely why certain structures were put in place or how and why the architecture patterns used were modified.

The structure of the rest of the paper is as follows: Section 2 lays groundwork by describing patterns, quality attributes, and tactics. Section 3 presents a general model of the interaction of patterns and tactics. Section 4 explores how patterns and tactics interact. Section 5 shows how to make the interaction of tactics and patterns visible in architecture documentation, through architecture diagram annotations. Section 6 contains several case studies where the annotation method has been used. Sections 7 and 8 describe related and future work.

2. Background: patterns, quality attributes, and tactics

2.1. Architecture patterns

Patterns are solutions to recurring problems. A software pattern describes a problem and the context of the problem, and an associated generic solution to the problem. The best known software patterns describe solutions to object-oriented design problems (Gamma et al., 1995), but patterns have been used in many aspects of software design, coding, and development. Architecture patterns describe an abstract, high-level system structure and its associated behavior (Maranzano et al., 2005). Architecture patterns generally dictate a particular high-level, modular system decomposition. Numerous patterns have been written for software architecture, and can be used in numerous software architecture methods (Harrison and Avgeriou, 2007b; Bosch, 2000; Jansen and Bosch, 2005; Shaw and Garlan, 1996).

An important advantage of patterns is that the consequences of using the architecture pattern are part of the pattern. The result of applying a pattern is usually documented as "consequences" or "resulting context" and is generally labeled as positive ("benefits") or negative ("liabilities"). Many of the benefits and liabilities concern quality attributes; for example, a benefit of the Layers pattern states, "Standardized interfaces between layers usually confine the effect of code changes to the layer that is changed . . . This supports the portability of a system." A liability of the same pattern states, "A layered architecture is usually less efficient than, say, a monolithic structure of a 'sea of objects'." (Buschmann et al., 1996) However,

the information is incomplete; it does not cover all the well-known quality attributes.

In addition, this information is currently inadequate. The impact of a pattern on a quality attribute is often given simply as a positive or negative impact (Harrison and Avgeriou, 2008a; Cloutier, 2006), but how that maps to a real system is not explained. In addition, architecture patterns are general solutions, thus their impact on quality attributes largely depends on how the pattern is used in the architectural design. Even more detailed elaborations of the impact, such as given in the excerpt above, currently do not explain how impact on quality attributes may be mitigated, or how the architecture can be changed to achieve the quality attribute of interest.

2.2. Quality attributes and tactics

Quality attributes are characteristics that the system has, as opposed to what the system does (Sommerville, 2007), such as usability, maintainability, performance, and reliability. Quality attributes are not simply met or not met, but rather, satisfaction is along a scale, always viewed within the specific context of a scenario: given a system state and specific input, the output is required to extend within specific limits (Bass et al., 2003; Bachmann et al., 2005). A particular challenge of quality attributes is that because they tend to be system-wide characteristics, system-wide approaches are needed to satisfy them; these approaches are defined at the system architecture level and not the component level.

Typically, systems have multiple important quality attributes, and decisions made to satisfy a particular quality attribute may affect another quality attribute. For example, decisions to maximize processing speed may come at the cost of additional memory needed. Therefore, architects must make tradeoff decisions: whether to implement software in a way that optimizes one quality attribute to the detriment of another. One approach for analysis of tradeoffs is given in (Bachmann et al., 2005). The Architecture Tradeoff Analysis Method (ATAM) is an architecture evaluation method that focuses on achievement of quality goals (Bass et al., 2003; Clements et al., 2002c). Steps include prioritizing quality attribute scenarios, and analyzing architectural approaches. The Cost Benefit Analysis Method (CBAM) (Bass et al., 2003) builds on ATAM to include business goals and economic issues. It develops architectural strategies that correspond to quality attribute scenarios and makes tradeoffs based on their costs and benefits.

Tactics are measures taken to improve quality attributes (Bass et al., 2003). For example, a duplication scheme to improve the reliability (nonstop operation) of the system is a tactic. Tactics impact the architecture patterns in various ways. In some cases, a tactic may be easily implemented using the same structures (and compatible behavior) as a particular architecture pattern. On the other hand, a tactic may require significant changes to structure and behavior of the pattern, or may require entirely new structures and behavior. In this case, implementation of the tactic, and future maintenance of the system, are considerably more difficult and error-prone. Tactics may be "design time," or overall approaches to design and implementation, such as "hide information" to improve modifiability, or may be "runtime tactics," which are features directed at a particular aspect of a quality attribute, such as "authenticate users" to improve security. In this paper we concern ourselves only with runtime tactics.

Because of the importance of quality attributes, it is critical that they be considered during early design; during system architecture. Indeed, we find that architects often consider them simultaneously (Harrison and Avgeriou, 2007b; Harrison et al., 2006). However, unless the relationship between the architecture patterns and the tactics being used is well understood, architects risk making architectural decisions (either concerning which patterns to use or which tactics to use) that could be difficult to implement correctly and maintain. Unfortunately, as noted above, the information about the relationship between architecture patterns and tactics is not adequate.

2.3. Patterns and tactics

There is a close relationship between architecture patterns and tactics. Bass et al. explain that tactics are "architectural building blocks" from which architecture patterns are created; that patterns package tactics (Bass et al., 2003). They also illustrate through example how the selection of tactics helps determine the selection of the architecture patterns used, which then leads to further selection of tactics. Thus we see that patterns, while they may be largely composed from tactics, also provide the starting point for incorporation of tactics. For example, a case study of the use of Attribute-Driven-Design (Wood, 2007) shows how the client–server pattern is selected very early, in the architecture. One of the reasons for the selection of the early structure is to help satisfy reliability requirements by providing a persistent storage service with state information. Subsequently, tactics are incorporated into the structure provided by the pattern that further satisfy reliability and availability requirements through duplication and other tactics.

As noted earlier, the selection of architecture patterns to use is taken very early. Of course, it encompasses various tactical decisions as well, but many others are yet to come. Bass et all put it nicely: "... in relating tactics to patterns the architect's task has only just begun when the tactics are chosen. Any design uses multiple tactics, and understanding what attributes are achieved by them, what their side effects are, and the risks of not choosing other tactics is essential to architecture design." (Bass et al., 2003, p. 126)

Of course, not all software development is greenfield; it is often necessary to enhance quality attributes of existing systems. In fact, this is probably much more common than designing a new system. This means that tactics must be added to the existing architecture patterns, regardless of how difficult it is. Knowledge of the interaction of patterns and tactics helps the system maintainer understand how to implement a given tactic in a given architecture.

3. A model of patterns, quality attributes, and tactics

This model shows in general terms how patterns, quality attributes, and tactics are related to each other, and how they are related to the overall architecture. It provides the framework for discussion of the detailed ways that tactic implementations affect the patterns used. It also provides a foundation for the description of a method of annotating architecture diagrams in a way that shows the impact of implementing tactics on the architecture (Fig. 1).

The model is comprised of entities associated with a system's architecture with respect to patterns and tactics; these may represent structures and behavior, as both patterns and tactics have structural and behavioral parts. We discuss each of the entities in the figure in turn. Let us begin with the architecture. An architecture consists of architecture pattern implementations in order to satisfy architectural concerns. (The architecture may also contain structures that are not part of any pattern implementation, though they do not concern this discussion.)

An architectural concern is a requirement on the system to be built, such as a feature to be implemented (IEEE, 2000; Hofmeister et al., 2005). A special type of architectural concern is a quality attribute. Quality attributes are important architectural concerns, and can have profound impact on the overall structure of the architecture. However, quality attributes are often the difficult architectural concerns to satisfy. Whereas architectural partitions

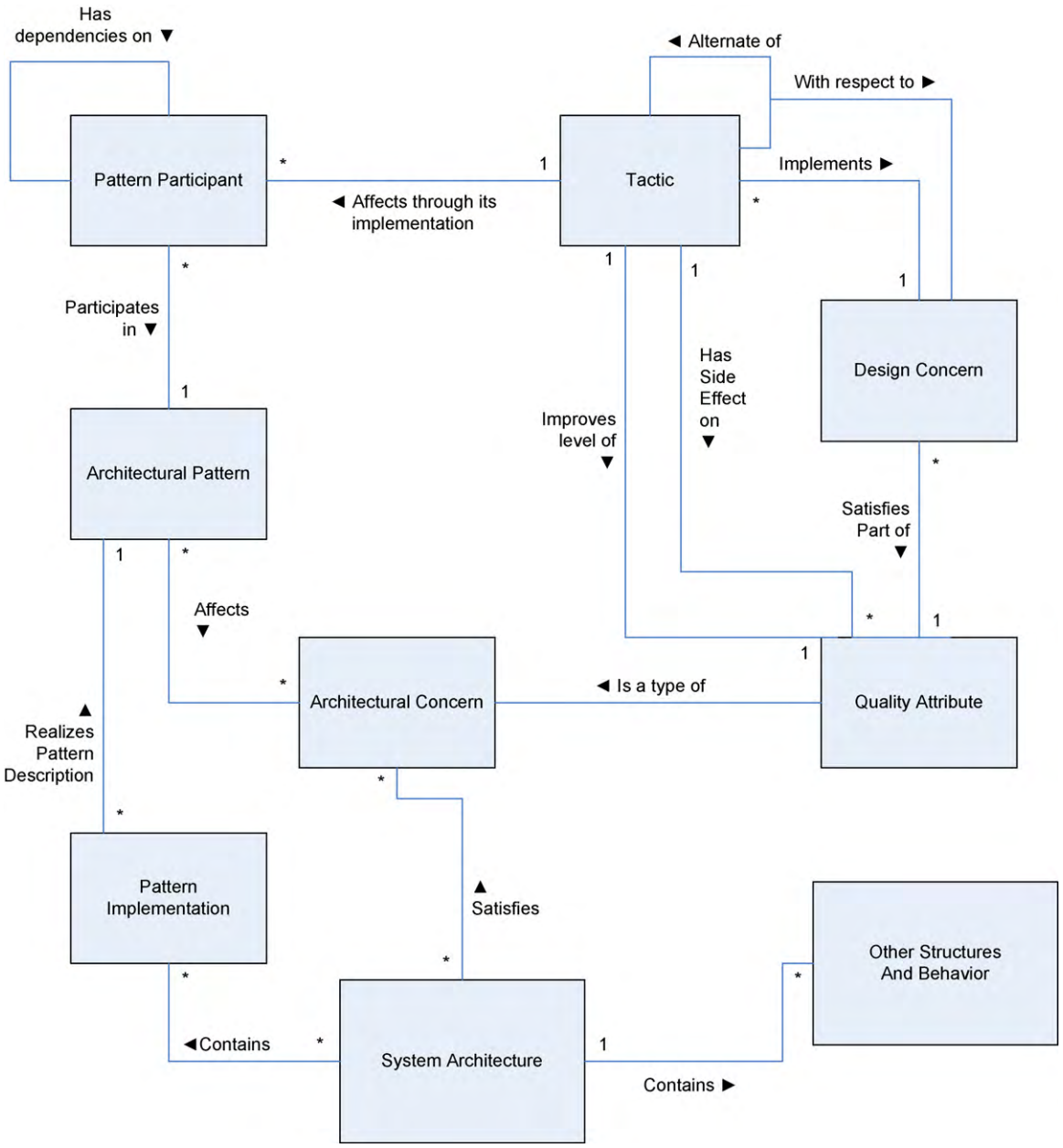

Fig. 1. Pattern, quality attribute, and tactic model.

(see patterns below) often readily fit functional requirements, quality attributes are system wide, and are satisfied through measures that cut across such partitioning. Quality attribute scenarios are often used to specify requirements on quality attributes (Bass et al., 2003).

Tactics are measures implemented to improve quality attributes. Tactics that do the same thing are grouped into categories, also called design concerns in ADD (see Wood, 2007.) These design concerns are selected following the quality attribute scenarios. For example, security can be improved by resisting attacks, detecting attacks, and recovering from attacks. These are design concerns (or categories of tactics) for security. The tactics within each category are implementations of the category. Tactics for the "resisting attacks" design concern include the following: Authenticate users, Authorize Users, Maintain Data Confidentiality, Maintain Integrity, Limit Exposure, and Limit Access.

In some cases, tactics are alternate ways of implementing a design concern. For example, a design concern for availability (often called reliability) is "Fault Detection." Two tactics for fault detection are "Ping/Echo" and "Heartbeat (dead man timer)" (Bass et al., 2003). We note that while the model applies to both design time and runtime tactics (see Bass et al., 2003), we have focused primarily on runtime tactics for simplicity.

The implementation of tactics improves the level of its quality attribute. For example, the tactic "authentication" improves the level of the Security quality attribute. However, tactics may also have side effects on other quality attributes. These side effects are usually negative.

The left side of the figure shows architecture patterns and their relations. An architecture pattern is a solution to an architectural problem, which is usually described as a set of architectural concerns or forces. The pattern satisfies multiple architectural concerns, but may also have side effects on other architectural concerns. A pattern may have different variants, and in fact, the variant used is often based on the tactics used (Bachmann et al., 2007).

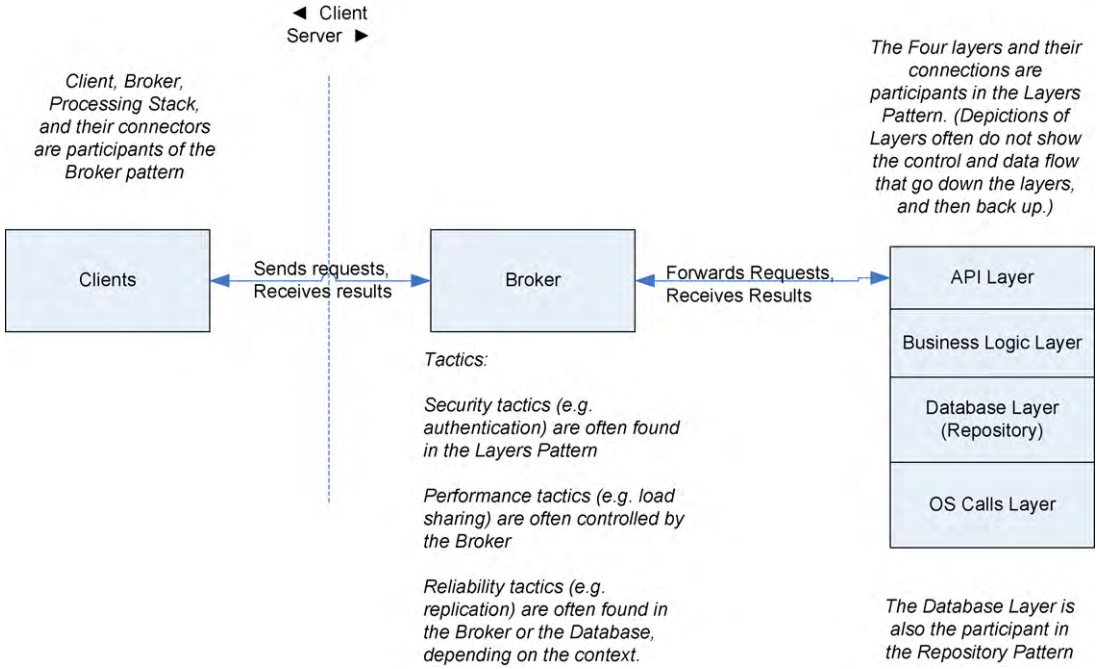

Fig. 2. Pattern participants in a typical diagram of an e-commerce system (simplified).

The solution of architecture patterns consists of components and connectors between them; together, these comprise the pattern participants. The pattern participants are interrelated; they depend on each other. The pattern participants have both a structure and a behavior. The pattern provides a generic solution as a collaboration among the pattern participants. The pattern implementation is a concrete realization of a pattern, including reification of all its components and connectors.

The relationship between the pattern participants and the tactics is significant. The code used to implement tactics can reside entirely within the pattern participants, in which case the pattern remains the same. However, implementing tactics often requires changes to the pattern participants: in some cases, it is merely necessary to replicate a pattern participant. In other cases, the structure and behavior of a participant must be significantly modified, or even new structures and behaviors added which are not part of the original pattern. We describe these changes in detail in later sections.

For example, consider a simple e-commerce application, consisting of multiple web clients that access a central server. (For simplicity, we give only parts of its architecture.) Its *Architecture* contains *Pattern Implementations* of the following *Architecture Patterns*: Broker, Layers, and Repository. Each pattern consists of *Pattern Participants*, The following figure is a simplified but common diagram of such an architecture. It shows the pattern participants, but as is typical in architecture diagrams, it does not explicitly show tactics. Comments in italics describe the pattern participants and discuss tactics (Fig. 2).

The *Architectural Concerns* include the features, the hardware and software platforms to be supported, security requirements, reliability requirements, etc. Of these, security and reliability are *Quality Attributes*.

A *Design Concern* used to satisfy part of the security *Quality Attribute* is "resist attacks" (Bass et al., 2003), and a *Tactic* to implement it is "authenticate users." This tactic is often implemented by adding a Layers' *Pattern Participant*; namely an authentication layer above all other layers.

4. Interaction of patterns and tactics

As noted above, quality attributes are addressed through the implementation of tactics. The implementation of tactics is done within the architectural structure of the system, including the structures of the architecture patterns used. Because this interaction can be very important, we study it in depth in this section. In an expansion of the upper part of Fig. 1, we see that pattern participants consist of components and connectors, and that tactics impact each in various ways, which are described below. In Fig. 3, the boxes represent instances of components, connectors, pattern participants, or tactics. For simplicity, we have omitted other entities usually employed in Architecture Description Languages such as ports, roles, and bindings; they are out of the scope of this paper but an interesting subject of future study.

In this portion of the model, we see that quality attributes are improved through the implementation of runtime tactics. The tactics are implemented in the participants (generally the components and connectors) of the patterns used in the architecture of the system. Note that connectors should be considered first class entities (Shaw, 1996).

We are particularly interested in the relationship between the tactics and the pattern participants. This is a key relationship, because the tactic being implemented may or may not be readily implemented, depending on what the pattern participants are, and how they depend on each other. We explore this relationship in depth below.

Patterns can impact individual tactics by making it easier or more difficult to implement them.

As described previously, quality attributes are satisfied through the implementation of tactics. Because tactics are implemented in the code, their implementation can be easier or more difficult depending on the structure and behavior of the system they are implemented in – its architecture. In fact, the structure and behavior of a tactic ranges from highly compatible with the structure and behavior of a pattern to almost completely incompatible with them. Naturally, the closer the

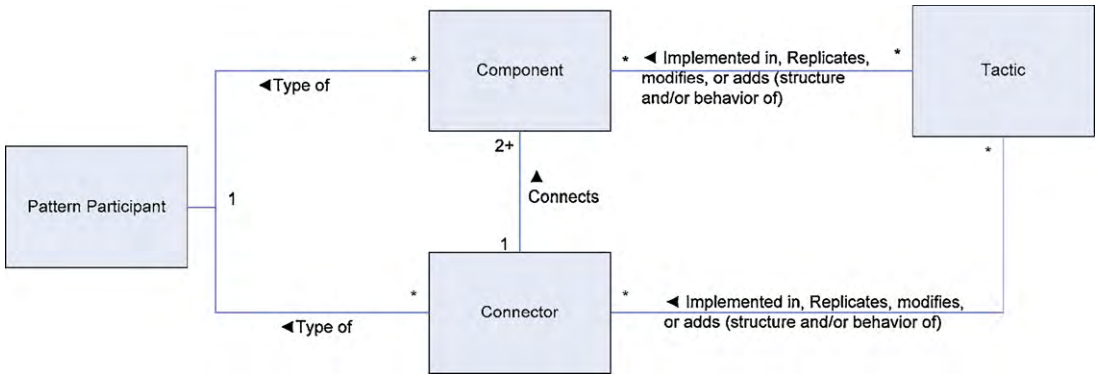

Fig. 3. Interaction of patterns and tactics.

Table 1
Changes to pattern components.

Type of structure change	Description	Impact
Implemented in	The tactic is implemented within a component of the pattern. The external interface of the component does not have to change. (This is a special case of the Modify type, described below.)	Only a single component must change, and in a minor fashion. This is the easiest type of change to implement.
Replicates	A component is duplicated. This is usually done to implement redundancy, for reliability, or to implement concurrency for performance. (This is a degenerate form of the "Add, in the pattern" type of change.)	Small changes are often required in the duplicated component, as well as components that interface to it. These changes are still usually easy to implement.
Add, in the pattern	A new instance of a component is added to the architecture. The component follows the structure of the architecture pattern, thus maintaining the integrity of the architecture pattern.	The component must be written, of course, but the changes required by the interfacing components are easy and well defined. These changes are easy or moderately easy to implement.
Add, out of the pattern	A new component is added to the architecture which does not follow the structure of the pattern. This causes the architecture of the system to deviate from the intent of the pattern	In addition to adding the new component, other components must change, often in significant ways. This is not only difficult to implement, the changed structure can make future maintenance more difficult
Modify	A component's structure changes (as opposed to adding new components in or out of the pattern.)	Such changes generally mean that other components must also change. The changes can range from minor to very difficult.
Delete	A component is removed	We have never observed it. One might consider the selection of a simpler variant of a pattern some sort of a delete; for example, using a variant of MVC that combines the model and view.

structures match, the easier it is to implement the task in the pattern.

The structure and behavior of a system are obviously intertwined, but are generally discussed and documented separately. Likewise, we will discuss the interaction of patterns' and tactics' structures, followed by a discussion of the interaction of their behaviors. We then describe the magnitude of the interactions, and end this section by showing an example of tactic data for a quality attribute and a few patterns. The example gives data chiefly about the impact of the tactics on the structure of the patterns.

4.1. Interaction of structure in patterns and tactics

We have identified several types of changes that a pattern's structure might undergo when a tactic is implemented, as shown in Table 1. The information may be used to modify the pattern, or if one has not selected an initial pattern yet, it might be used to understand the tradeoffs of selecting one pattern versus another. The types of structure change are listed in general order of amount of impact on the structure of the pattern; that is, the earlier types generally have less impact than the later types. However, the amount of impact depends on the pattern and the tactic. In particular, modification of a component has a large range of potential impact. Each pattern/tactic pair must be individually examined to determine the impact. Magnitude of impact will be discussed after the discussion of behavior.

In addition to pattern components, the connectors between the components are affected. It is certainly the case that changes to the components may cause corresponding changes in the associated connectors' ports, interfaces, etc. The table below shows changes to components, followed by a table showing the changes to the connectors (Table 2).

We show examples of each of the changes below. Short descriptions of the patterns and tactics used here can be found in Appendix A.

Table 2
Changes to pattern connectors.

Type of change to a component	Corresponding change to connectors
Implemented in	No change.
Replicates	Connectors added between replicated components and other components. They may be within the structure of the pattern.
Add, in the pattern	New connectors added within the pattern structure.
Add, out of the pattern	New connectors added outside the pattern structure.
Modify	New or modified connectors needed, probably outside the pattern structure.

1. Implemented in: Implementing the Ping/Echo reliability tactic in a Broker architecture: the Ping/Echo tactic has two major components, the component issuing the ping and keeping the time, and the components responding to the ping messages with an echo. Since the Broker component already maintains communication to all the server components, the Broker becomes the component that issues the ping. The server components simply respond to it. This requires a small change to the messaging protocol (the connector), but the change can be lessened further. The work requests to the server components can double as the ping messages, and their responses can be the echo messages. In this case, only minor changes are required in the broker component, and the other components have few if any changes.
2. Replicates: In a client–server architecture, improved performance is often needed, so the Introduce Concurrency tactic is used. It simply duplicates the server component. A connector between the servers may be added to allow the duplicated servers communicate and share load. This configuration is often used in web-based download servers that have local mirror sites.
3. Add, in the pattern: Security of a layered system can be enhanced by adding the Authentication tactic as an additional layer on top of everything. The security layer is a new component and associated connector, but it is within the structure of the Layers architecture. Changes to the other parts of the system will likely be confined to the (former) top layer of the system.
4. Add, out of the pattern: Consider a Pipes and Filters-based system that needs to add the Ping/Echo reliability tactic. Ping/Echo requires a component to manage the pinging, and to know who does not answer. However, Pipes and Filters has no possible candidate components, so one must be added. Furthermore, each filter must now respond – in good time – to the ping messages, requiring new connectors. This may mean that each filter must implement an interrupt handling mechanism. The new architecture looks quite a bit different from the pure Pipes and Filters architecture.
5. Modify: Consider a Blackboard architecture. The reliability tactic Checkpoint and Rollback is to be implemented. Because the behavior of processing in a Blackboard system is somewhat emergent, it will be very difficult to define checkpoints to which one can roll back. Defining such checkpoints is counter to the very idea of a Blackboard. An alternative might be to log all inputs in order, so one could presumably reconstruct the sequence of events, but this could be cumbersome, and require significant storage. In any case, the blackboard component must undergo significant changes, though no additional component is added. Changes to connectors would be minor or even unnecessary.

4.2. Interaction of behavior in patterns and tactics

Within behavior, one can consider the actions of the software, and the state transitions of the software. We will focus on the actions because they are more general: every state transition is accomplished by an action. Within actions, we see two notable issues:

1. Sequences of actions: Sequences of actions are often represented in message sequence diagrams, particularly for messages between different components.
2. Timing of actions: Some actions must be completed within a certain time limit. Different actions are taken if the action does not finish within its allotted time. (In some cases, such as hard real time systems, the integrity of the system is compromised if actions do not complete on time.)

We discuss each below.

Table 3
Structure and behavior changes in pattern participants.

Type of structure change	Typical behavior change	Impact
Implemented in	Actions are added within the sequence of the component. Timing changes are mainly limited to the component.	This type of behavior change is easy to implement. Timing of the component is well defined. The timing of the component can be modeled or compared to the previous version, which can be used to assess the impact on system realtime requirements.
Replicates	When a component is replicated, its sequence of actions is copied intact, most likely to different hardware. Ideally, their behavior does not change at all. Whatever component that controls the replicated component requires actions added, normally within its sequence.	Changes in the actions are limited to the controlling components, and as such are quite easy to implement. In unusual cases, the parallelism of replicated components can slightly improve timing, if it is properly exploited.
Add, in the pattern	The new component added comes with its own behavior. In most cases, the behavior will have to follow the constraints of the pattern; the new component's actions are within the message sequence of the pattern.	This action is moderate work, because new actions must be developed. Timing changes are mainly limited to the added component.
Add, out of the pattern	The component added will have its own behavior. The actions do not have to follow the pattern; in fact, they may be an action sequence that is independent of the sequences in the pattern.	The new component does not have to work within the constraints of the pattern's action sequences, which may make the addition of the component easier. However, every point where the new component interacts with the existing components must somehow deal with asynchronous actions of the new component. This implies, for example, interrupt processing. This is likely significant work.
Modify	The change to the structure of a component implies changes or additions within the action sequence of the component that are more significant than those found in "Implemented in." It is conceivable but unlikely that any actions are added independent of the component's action sequence. (Note that the magnitude of modifications can vary widely; in some cases "Modify" is lower impact than "Add out of the pattern.)	The magnitude of the work needed depends on the situation. However, note that interfaces of the component may change, which means that the behavior of other components must change correspondingly.

4.2.1. Adding action sequences

Most of the behavior of architecture patterns is characterized as action sequences. For many common patterns, these sequences can be found depicted in message sequence charts (Maranzano et al., 2005). These charts depict the essential common sequences of actions in systems using these patterns. Tactics also have behavior, namely the sequence of actions that satisfy the quality attribute concern. The behavior of tactics can change the behavior of the pattern by adding or modifying action sequences of the tactic to those of the pattern.

The following are examples:

1. Adding the security tactic "authenticate users" adds a sequence of actions at the beginning of the action sequence that initiates an interactive user session.
2. Adding the "Ping/Echo" reliability tactic adds a sequence of action that is controlled by a clock, independent of any other actions in the system. It also adds timing behavior: sending the pings is controlled by time, and the responses must come within a certain time or the tardy component is considered to be dead.

Behavior changes are associated with structural changes. Let us revisit the types of structural changes, and note the types of behavior changes that are associated with them.

They are as follows, given in general order of increasing impact (Table 3).

Let us look at some examples:

1. Implemented in: The Ping/Echo reliability tactic can be added within the structure of the Broker pattern. The behavior of the tactic is to periodically send ping messages to selected components, and they reply. The broker's action sequence is to wait for requests from clients, and distribute the requests to appropriate servers. To implement Ping/Echo, the broker can use the clock to decide when to send ping messages, just as it sends client requests out. It must add behavior to keep track of responses. The servers must add behavior to respond to ping messages right away.
2. Replicates: Consider the Hot Spare tactic implemented in the Broker pattern. A server is replicated, and the broker controls it. The servers have no changes in their actions. Sequences of actions are added to the broker as follows: broker sends requests to two servers. It must also manage them, perhaps using Ping/Echo to determine their health. These are all additions within the broker's normal sequence of actions.
3. Add, in the pattern: consider the addition of the Authentication security tactic to the Layers pattern. An additional layer for authentication is added on top of all the other layers. It adds behavior to handle the authentication, which must come before other actions in the system. For example, requests for service must pass through the authentication layer to ensure that the requestor is authenticated, then the normal sequence of actions proceeds.
4. Another example of addition within the pattern is adding the Encryption tactic to the Pipes and Filters pattern. This is interesting to consider because there are two obvious ways encryption might be added. In the first way, all the filters might operate within a secure environment, and encryption is applied to the final product. In this case, a filter containing the encryption actions is added to the end of the Pipes and filters configuration. The behavior of the other filters is not affected. In the other case, the filters are distributed across an insecure network, and the results of one filter must be encrypted before the data is passed to another filter. In this case, each filter must add another filter for encryption after it. But like the first case, the added actions are within the sequence of actions – encryption must follow the actions of each filter. By the way, this implementation also requires that decryption be done in a filter before each filter – each filter becomes three!
5. Add, out of the pattern: Consider adding the security tactic, Intrusion Detection to the Layers pattern. Intrusion Detection runs as an independent process that monitors the open ports on a system for attempts to intrude. It is added as a component out of the pattern. Its actions are completely independent of the application. It makes it straightforward to implement. But what should it do when it detects intrusion attempts? Should it raise a warning immediately? If so, the Layers system must handle interrupts from it. Should it simply log the events? If so, the Layers system needs to have behavior added to occasionally check the log files and clean them out. In this case, the actions can be part of the Layers' normal sequence of actions.
6. Modify: Consider a Layers architecture in which the reliability tactic Passive Redundancy (warm spare) is implemented. Of course there is replication of the most of the system (all except the top layer, perhaps). Then behavior must be added to a certain layer to send updates to the warm spare. This requires more substantial action additions, because the system must determine what state information to send, and the spare system must be modified to replicate the active's state through the state messages. Although it is likely that these additions are all within the existing action sequences, the changes are significant.

4.2.2. Time behavior

It is important to consider not only actions but timing of actions. Obviously, the addition of any actions causes the timing of the system to change in some way. Because functionality is being added, in most cases, the timing change means that the overall processing can take longer. (Exceptions include adding parallel processing and other performance-related tactics.) The timing change is often not important. In many cases, changes in the timing of the system or component are small enough that the timing change can easily be ignored. However, if there are any critical timing constraints in any part of the system, the impact of implementing the tactic on the timing of the system must be examined. We discuss timing in detail in a later section.

The timing changes caused by the implementation of a tactic may or may not be of concern, depending on the nature of the tactic and the timing constraints of the system. Timing can be a system-wide issue – just because a component's action sequence is left unchanged by a tactic, one cannot automatically assume that the timing of that component is not compromised. If different components share the same processor, changes in the actions of one can impact the timing of the others.

Tactics impact the timing of the system in two main ways. First, the tactic may add new timing constraints to the system. Second, the tactic may change the timing of the system in a meaningful way (i.e., change the timing enough that we care.) Each of these ways can be either explicit or implicit. We explain each in more detail below.

1. Adding new timing, explicit: In this case, the tactic contains specific timing requirements in its specification. The most obvious example is the Ping/Echo reliability tactic. In this tactic, a central control component sends out sanity pings, and requires that other components respond within a certain time. Note that although the timing itself is implemented in a central component, timing requirements extend to each of the components being monitored: each component must respond within a certain time. Therefore, the time behavior of all the responding components must be examined to ensure that the timing

requirements of the echo message can be met, and that the time behavior of the component itself is not compromised by servicing the ping messages.
2. Adding new timing, implicit: In this case, the tactic does not contain specific timing requirements. However, certain timing constraints can be inferred in the behavior of the tactic. For example, the Passive Redundancy (Warm Standby) reliability tactic has implicit constraints. In this tactic, a standby component receives regular state updates from the active. There is an implicit requirement that the warm standby be able to process each state update message before the next one comes. Otherwise, the state messages would either back up or would be discarded, or would force the active to wait until the standby catches up. The first two would cause loss of synchronization between the active and the standby, and the third would cause performance problems. Therefore, the standby must process state messages fast enough to keep up.
3. Changing Existing Timing, Explicit: These are tactics that explicitly change the performance of the system. In general, they add no new behavior. All the performance tactics (processing speed) intend to change the timing of the system. For example, the tactic Add Concurrency improves the timing of the system. Naturally, there are no tactics that explicitly (i.e., purposely) degrade the timing of the system.
4. Changing Existing Timing, Implicit: This case is where a tactic changes the timing of the system as a side effect; there is no intent to do so. In nearly all cases, the change is to cause processing to take longer, although in many cases, the timing change is too small to be of concern. Virtually all tactics introduce some change in timing., though most are small. An interesting example is the Voting reliability tactic. In this tactic, different components work on the same problem. Their results are compared, and a voting scheme is used in the cases where the results differ. The implicit timing change is that all the voter components must complete their tasks before further progress can be made. Thus the performance is limited by the slowest voter.

4.3. Magnitude of impact

The sections above describe the ways in which structure and behavior of the patterns are changed by the implementation of tactics. An important consideration is the size of the impact. The size of the impact – how much a pattern must change – is an indicator of the work required to implement the tactic, how error-prone the implementation might be, and how difficult it might be to see the pattern in the system at a later time. However, although one can get a general sense of the magnitude of the impact of types of changes, it can vary depending on the particular pattern and tactic. Therefore, it is necessary to examine each pattern-tactic combination.

We defined a five-point scale to describe how difficult it is to implement a particular tactic in a given pattern. It is based on the impact on the structure and behavior of the pattern. The descriptions, expanded from (Harrison and Avgeriou, 2008b), follow:

1. Good Fit (+ +): The structure of the pattern is highly compatible with the structural needs of the tactic. Most or all of the changes required are the "Implemented in" type, and the behavior of the pattern and tactic are compatible. Any structure changes ("Modify") required are very minor. For example, the Broker architecture strongly supports the Ping–Echo tactic because the broker component already communicates with other components, and is a natural controller for the ping messages.
2. Minor Changes (+): The tactic can be implemented with few changes to the structure of the pattern, which are minor and more importantly, are consistent with the pattern. Behavior changes are minor, and generally encapsulated within pattern participants (e.g., no new message sequences between participants added.) These types of changes are "Replicates" or "Add, in pattern." Structure changes ("Modify") are minor. For example, the Layers pattern supports the active redundancy tactic by replicating the layers, and adding a small distribution layer on top. Although another layer is added, it is entirely consistent with the pattern.
3. Neutral (∼): The pattern and the tactic are basically orthogonal. The tactic is implemented independently of the pattern, and receives neither help nor hindrance from it. For example, consider implementing the reliability tactic "Ping/Echo" in the Model View Controller (MVC) pattern: there is no inherent conflict between them. But MVC has no natural place for handling them either, such as an exception handling layer one might implement in the Layers pattern. Therefore, MVC neither helps nor hinders the implementation of Ping/Echo.
4. Significant Changes (−): The changes needed are more significant. They may consist of "Implemented in," "Replicates," and "Add in pattern" where behavior changes are substantial. More often, they include significant "Modify" or minor "Add out of pattern" changes. For example, the Presentation Abstraction Control manages simultaneous user sessions. Implementing a Rollback tactic would likely require significant extra code to ensure that different interfaces are synchronized.
5. Poor Fit (− −): Significant changes are required to the pattern in order to implement the tactic. These consist of significant "Modify" and/or "Add out of pattern" changes. The structure of the pattern begins to be obscured. Adding the tactic requires the addition of behavior that is different from the original behavior of the pattern. For example, introducing Ping–Echo into a Pipes and Filters pattern requires a new central controlling component, along with the capability in each filter component to respond to the ping in a timely manner.

While the above list gives an overview of the levels of magnitude of impact, a key determiner of magnitude is the number of pattern participants that must change. Clearly, as the number of participants that change increases, effort for implementation, maintenance, and understanding increase. And the opportunities for errors increase.

Most changes involve the addition of a component (namely replication or adding in or outside the pattern; see above.) Typically, this requires adding a corresponding connector, and changing an existing component that communicates with the component. Thus we see that three changes becomes a natural "break point," for impact. The same applies to the "Modify" change: a change to a component typically requires a corresponding change to its connector as well as the component it communicates with. "Implemented in" is a minor change to a single component. This gives the following table of magnitude for each type of change (Table 4).

It is not uncommon for a tactic to involve more than one type of change; for example, add in the pattern (adding a new component and connector) and Modify (an existing component.) In such a case, one would consider the upper and lower ranges of both types of changes. In this example, add in the pattern (2 changes) has a range of ++ to +, and Modify (1 change) has a range of ++ to −. This gives a total range of ++ to −, but it is biased toward the positive end; the modification required is likely minor.

A particular consideration is the case where a tactic aligns with a previous tactical decision. Within the meta-model, two tactics would affect the same pattern participants in the same way; they have overlapping impacts. Obviously, the effort to implement the tactic is much less than if it were implemented in the pattern without the previous tactic. We can model the difficulty of the second tactic by considering that it is implemented in the architecture as it now stands – including the previously implemented tactic. This

Table 4
Impact magnitude as a function of number of participants impacted.

Change type	Number of changes	Impact range	Comments
Implemented in	1	++ to +	Should impact only a single participant.
Replicates	3 or less	++ to +	
	More than 3	+ to ∼	
Add, in the pattern	3 or less	++ to +	
	More than 3	+ to ∼	
Add, out of the pattern	3 or less	∼ to −	
	More than 3	− to − −	
Modify	3 or less	++ to −	The wider range indicates inherent variability of modify.
	More than 3	∼ to − −	

could, for example, result in the impact on a pattern participant being "Implemented in" or "add, in the pattern", rather than "add, outside the pattern." Case Study 1 shows such an example. This begs the question of whether the order of tactic implementation is significant; we address this in Section 7.

It is important to note that the architecture of every system is different, and has different requirements and constraints, resulting in a unique structure. We see this reflected in the ranges given in the above table. Therefore, the impact of a tactic on an architecture could be different from our findings. As such, these findings should be taken as general guidelines. They are most useful as guides for comparisons of tactics and of patterns, and as helps for implementation.

Note that while we consider impact on development effort, there is another dimension of impact; namely that a tactic may impact other quality attributes; e.g., tactical decisions taken to improve security often have a detrimental effect on performance. Such tradeoffs must also be taken into account, and are also part of the architectural tradeoff analysis one must do (Bass et al., 2003).

4.4. Impact data example: reliability

This example describes the impact of several common reliability tactics on one pattern, the Pipes and Filters pattern. Each paragraph begins with an assessment of the magnitude of the impact, followed by a description of the structural and behavioral issues of implementing the tactic in a Pipes and Filters architecture.

In order to be complete for reliability tactics, each common reliability tactic must be analyzed for each common architecture pattern. The analysis for one other pattern, Layers, is available in (Harrison and Avgeriou, 2008b), but there are many other common architecture patterns. In addition, other quality attributes have their own sets of common tactics.

4.4.1. Pipes and filters

Summary: The Pipes and Filters pattern is chiefly used for sequential actions such as processing data streams or incremental processing. Its hallmarks are performance and flexibility, but it is often also found in highly reliable systems. It has a few significant liabilities that must be overcome in such systems.

1. Fault Detection
   a. Ping/Echo: − −: A central monitoring process must be added, which must communicate with each filter. Each filter must be modified to respond in a timely manner to the ping messages. This not only affects the structure of the pattern, but may conflict with realtime performance. (Add out of the pattern, along with moderate changes to each filter component.)
   b. Heartbeat: − −: Similar to Ping/Echo. It is a bit easier to add the heartbeat generating code to the filters, because they do not have to respond to an interrupt. However, each filter must still send the heartbeat in response to a timer. (Add out of the pattern, along with moderate changes to each filter component.)
   c. Exceptions: − −: The problem here is who should catch an exception when one is thrown. If the exception can be completely handled within a single filter component, then there is no problem. However, if the filter cannot fully handle the exception, who needs to know? Depending on the application, it may be the subsequent filter (simple modifications needed to the filters), or a central process might need to know (add out of the pattern, along with moderate changes to each filter component.)
2. Recovery – Preparation and Repair
   a. Voting: + +: To implement voting, create different filters as the voting components. Create the receiving component (the voter) as a filter. To distribute the input to the different voting filters, use a pipe that has one input and multiple outputs (e.g., the Unix "tee" command.) (Add in the pattern, but the work besides using different algorithms in the different filters is very straightforward; almost trivial.)
   b. Active Redundancy: + +: Replicate filter components. Send the same stimuli to redundant filters. Use a pipe or a final filter to receive the results from the redundant filters. You can arrange it so you take the first one finished, which improves performance; a common goal in Pipes and Filters. (Replicate, plus add a trivial pipe or filter to handle the results, as well as a distribution pipe, as noted in Voting. As in voting, the adds here are trivial.)
   c. Passive Redundancy: −: Replicate the filter you are backing up. Then modify the primary filter to send occasional updates to the backup filter. The backup filter must be modified to receive the updates, rather than the normal input data. A pipe or trivial filter is needed to handle the results, just as in Active Redundancy. This can be done within the pattern, but Active Redundancy is generally a superior tactic, and it fits so well, that this pattern is not recommended with Pipes and Filters. (Replicate, plus significant changes to the filters.)
   d. Spare: +: Set up a device as the spare, with the ability to run as any of the different filters. Create a new filter that handles distribution of work. It must detect when a filter does not respond to sending work (e.g., did the data write fail), and then initialize the spare as that kind of filter. (Add in the pattern, but the new filter is not trivial.)
3. Recovery – Reintroduction
   a. Shadow: +: In Pipes and Filters, Shadow is implemented similarly to Voting. In this case, the receiving component checks the results from the shadow against the results of the primary filter to see if they are correct. It may be necessary to communicate the state of the shadow filter back to the filter that distributes the work. Note that the shadow filter itself should need no changes. (Duplication with simple add in the pattern, plus possible small Modify to two filter components.)

   b. State Resynchronization: – –: The biggest problem with this tactic is filters should not have states except within processing of one piece of data. And in that case, it usually makes most sense to restart processing of that data from the beginning. If you must to implement this tactic, define states for each filter and create a mechanism to restore a filter to the proper state when it comes back up. That may require a monitoring process. (Major changes to components, plus possible add out of the pattern.)
   c. Rollback: – –: Check pointing is easy to do. However, once the data passes to the next filter, it is extremely hard to undo it – it is gone. If you must use it, use a monitoring process and a protocol of checkpoints to ensure the integrity of the data at the end of each filter. (Add out of the pattern, plus major changes to components.)
4. Prevention
   a. Removal from Service: –: Use a monitoring process to decide when to remove a filter from service. (Add out of the pattern. Minor changes may be needed for reconfiguration.)
   b. Transaction: ∼: Filters work naturally on streaming data, rather than transaction-oriented data. However, the first filter in a sequence might package the data into transactions, and later filters could operate on transactions of data (A trivial example not directly related to reliability used by one of the authors in a college course is calculating bowling scores with a Pipes and Filters architecture. The first filter packages the raw data into "frames", and the second filter sums the scores of each frame.)
   c. Process Monitor: –: Use a monitoring process to detect when a filter fails, and reconfigure the system. (Add out of the pattern. Minor changes may be needed for reconfiguration.)

5. Implementing tactics in patterns

How does one use the model presented above in the implementation of a system?

There are numerous processes for architectural design (Hofmeister et al., 2005), including some that explicitly involve architecture patterns (Harrison and Avgeriou, 2007b). Regardless of the process, one must weigh architectural alternatives, including which patterns and tactics to use. When one is considering which patterns and/or tactics to use, one needs to know:

1. Where in an architecture under consideration would a tactic be implemented?
2. What is the magnitude of the impact of a given tactic on a given architecture?

If a legacy system is under consideration, one also needs to know the implementation constraints when adding new tactics. For example, if the Passive Redundancy tactic is added in order to improve the reliability quality attribute, one needs to know the conditions and methods of data storage (including which repository architecture pattern is used – see Avgeriou and Zdun, 2005), and where it sits in the architecture (e.g., as a layer in the Layers pattern.).

Existing architecture methods deal with making decisions about these questions (see related work section for details.) In order to help them do so, we show a method of annotating architecture diagrams with tactic information. It can be used, for example, to annotate the logical view of a 4+1 architecture (see (Kruchten, 1995).) This annotation can be used to evaluate alternatives during initial architecture, and can be used to show tactic information of existing systems.

5.1. Documenting tactics in architectures

Because implementing a tactic in an architecture can significantly change the architecture patterns, it can be difficult to find the patterns in the architecture diagrams, making understanding of the system more difficult. This is particularly problematic in legacy systems, where implementing tactics is actually a form of "architecture drift" (Rosik et al., 2008). Therefore, it is important to be able to easily reflect tactics in architecture documents, including existing ones.

Comprehensive architecture documentation presents multiple views of the architecture (IEEE, 2000; Kruchten, 1995). A view is defined as a representation of a set of system elements and the relationships associated with them (Clements et al., 2002a). Clements et al. describe several different generic types of views, or viewtypes, which highlight different characteristics of the architecture (Clements et al., 2002a). Kruchten et al. also note different views of the architecture (Kruchten, 1995). An important view of the architecture is the components and connectors view. In spite of the fact that as Clements et al. point out, no single view can fully represent an architecture, the components and connectors view is the only architecture documentation of many systems. Informal diagrams may consist of lines and boxes, where boxes usually represent some sort of components or sub-components, and the lines between them represent some relationship between components. Other architecture documentation may consist of UML, or employ ADLs or other formalisms (Allen and Garlan, 1994; Allen, 1997; Shaw et al., 1995). However, the diagram styles vary widely; there is no standard that is universally used in architecture diagrams (Harrison and Avgeriou, 2008a).

These points lead to three requirements on a tactic documentation method:

1. In order to meet the needs of existing architectures, it should be possible to apply it to existing architecture diagrams. Furthermore, it should be clear what changes were made, and that the changes are associated with adding tactics.
2. It should be adaptable to differing amounts of formality in the existing description. Therefore, it should be semantic-free, but ideally should allow for adding semantics as desired by the user.
3. Any method of documenting tactics must be compatible with different architecture diagramming styles.
4. It must be easy to add to the diagrams. If it is difficult, it will not be used.

We propose a method of annotating architecture diagrams with tactic implementation information. It is a clear way to document the addition of tactics to architecture documents that is also easy to use, allows the architectural components to remain visible, and can annotate many different styles of diagrams. It is based on the types of changes to architecture patterns as described earlier.

There is a great deal of variation in styles of architecture diagrams. In our studies of 47 architecture diagrams (Harrison and Avgeriou, 2008a), we found that while all used boxes and lines, the meanings of the boxes and lines differed. Few explicitly used standard notations such as RUP or UML. Therefore, a key advantage of this annotation is that the annotation is not tied to any one notation. Regardless of the architecture notation style used, people can continue to use it and add the annotation to show the tactics added and changes required. This is especially true because the annotation is basically independent of the diagram semantics. One can add annotations to boxes or lines as needed. And other methods besides the circles may be used. For example, UML components can be annotated through stereotypes or notes instead.

Table 5
Sample quality attribute abbreviations.

R	Reliability
S	Security
U	Usability
PF	Performance
PO	Portability
M	Maintainability
CP	Capacity
CF	Configurability
E	Extensibility

Table 6
Sample tactic list.

ID	Tactic
S1	Authorization (security ++)
S2	Encryption (security ++)
R3	Ping/Echo (reliability − −)
PF4	Concurrency (performance ∼)

The flexibility of annotation is shown in the examples and in the case studies, which show annotations of diagrams with differing semantics and syntax.

5.1.1. Annotation style

The annotation method consists of two things: a list of tactics applied and circles showing the location and type of changes to the architecture for a tactic. The list of tactics consists of entries that associate an implemented tactic with the circle that show its implementation. Each entry in the tactic list consists of a unique identifier and the name of the tactic. The identifier consists an abbreviation of the quality attribute, followed by a number. The number uniquely identifies changes for that tactic. The name of the tactic is the name as shown in the examples above. Each entry in the tactic list also gives the magnitude, using the magnitude scale given earlier. An entry may also contain additional explanatory text, if desired (examples of such text can be seen in the case studies.)

A sample of quality attribute types follows. Note that there is some disagreement among experts as to the taxonomy of quality attributes. Therefore, this is a sample list only, based on quality attributes given in (International Standards Organization, 2001) (Table 5).

This is an example of a tactic list. The names in parentheses are for the reader's convenience; architects may or may not choose to include them (Table 6).

The types of changes to the pattern participants follow the types described earlier; namely, implemented in, replicated, added within the pattern, added outside the pattern, and modified. Each circle contains an abbreviation for the type of change. We use the abbreviations as in Table 7.

The circle has the tactic id at the top, and the type of change below, as Fig. 4.

Atypical circle looks like Fig. 5.

This shows an addition to a component within the constraints of the pattern, for the tactic listed as S1. If the previous table is being used, S1 indicates the Authorization tactic for security.

The circles are placed in the diagram on the pattern participant (e.g. component, connector, message) that is changed for that tactic. The circles should be placed on all the participants that change. (If placing a circle on a pattern participant detracts from the clarity of the diagram, i.e., obscures the writing in a box, it may be placed next to the participant.) Note that for a given tactic, the type of change may vary for different participants, so the change types may vary accordingly. (That is why each circle contains a change type.) Note that where tactics result in the addition of participants, the latter are added to the diagram (as one would normally do), and annotated with a circle.

Table 7
Types of change to pattern participants.

Abbreviation	Type
I	Implemented in
R	Replicated
AI	Added, in the pattern
AO	Added, out of the pattern
M	Modified

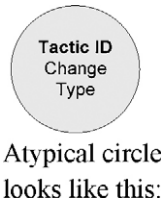

Fig. 4. Tactic implementation annotation style.

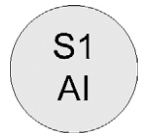

Fig. 5. Sample tactic implementation annotation.

This example shows the addition of the security tactic Authorization and the reliability tactic Ping/Echo to the Pipes and Filters architecture. Fig. 6 shows the Pipes and Filters architecture before the addition of the tactics. Authorization generally requires a single point of authorization, so a central authorization server is added to provide authorization (Fernandez and Ortega-Arjona, 2009). When a filter receives work, it first contacts the authorization server to receive permission for the work. We see that the authorization server and its connectors are added outside the pattern (abbreviation AO), while the filter components are modified (abbreviation M). The resulting architecture diagram shows authorization added (Fig. 7).

We now add the Ping/Echo Reliability tactic to this architecture. We can take advantage of the central authorization server to also manage the ping messages, and rename it the Central Control.

Ideally, one has both the "before" and "after" diagrams; that is, Figs. 6 and 8. However, the latest diagram should be sufficient to both identify the architectural structure and guide the implementation of the tactics.

The above example shows an annotated structure diagram, with annotations added to components and connectors. In a similar manner, annotations may be added to behavior diagrams. The same change types are used as for structure diagrams. In behavior they have similar meanings, as described previously.

Fig. 6. A simple Pipes and Filters architecture.

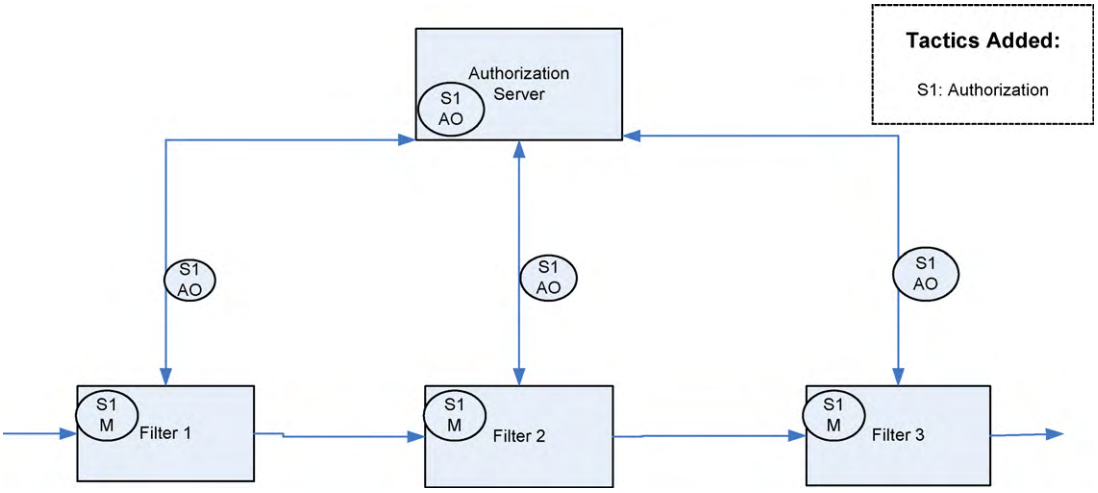

Fig. 7. Addition of authorization tactic to Pipes and Filters architecture. *Note*: In this diagram, the arrows on the connectors show possible direction of communication. The connectors to the authentication server show that the authentication server can respond to the filters.

Where an architecture has both structure and behavior diagrams, the same tactic identifiers should be used on both diagrams, so that one can easily recognize the impact of implementing a tactic on both the structure and behavior of the architecture.

The following example shows a message sequence chart for Broker pattern (from Buschmann et al., 1996) showing the addition of the Voting tactic, used to increase reliability (Bass et al., 2003).

In Fig. 9, we see that the Broker component modifies its "find servers" action to find multiple servers that will participate in the voting. It adds additional "call service" messages to go to the voting servers. When messages are received, the Broker adds behavior to wait until all voters have responded, and then select the proper response. We see that the behavior of the servers and the client do not change.

For simplicity, the diagram shows only two servers.

5.1.2. Examples

The following examples were taken from real architecture diagrams from (Booch, 2007). In each case, we added a tactic that would be a likely candidate for that application. These examples show how even with different architecture diagram styles, it is possible to annotate them with tactic information. We note that it is entirely possible that the tactics we chose are already implemented in the systems, but the architecture diagrams do not indicate it.

5.1.2.1. Google. This example shows the Google architecture, as described in (Brin and Page, 1998). We see the Shared Repository pattern in the upper right corner for Fig. 10; it is shared by at least the Indexer and the Store Server. The Store server also serves as a Broker of sorts for it. We choose to improve performance by adding the performance tactic, Add Concurrency. To do so, we replicate the Repository component and modify the Store Server component to handle the duplication. The Indexer component may need to change in a small way to handle multiple Repository components. We see in this example that the main change happens in one pattern (Repository), and an associated change happens in another pattern (Broker.)

5.1.2.2. Ambulance. Fig. 11 is from an ambulance information system, from (Stevens et al., 1998). The upper rectangle labeled "Statistics" is a Shared Repository. We wish to improve reliability by duplicating the repository, using the Passive Redundancy (Warm Spare) pattern. (Note: this improves the availability of the system by allowing the system to continue to operate if one of the databases is to fail; reliability of data integrity is clearly already supported

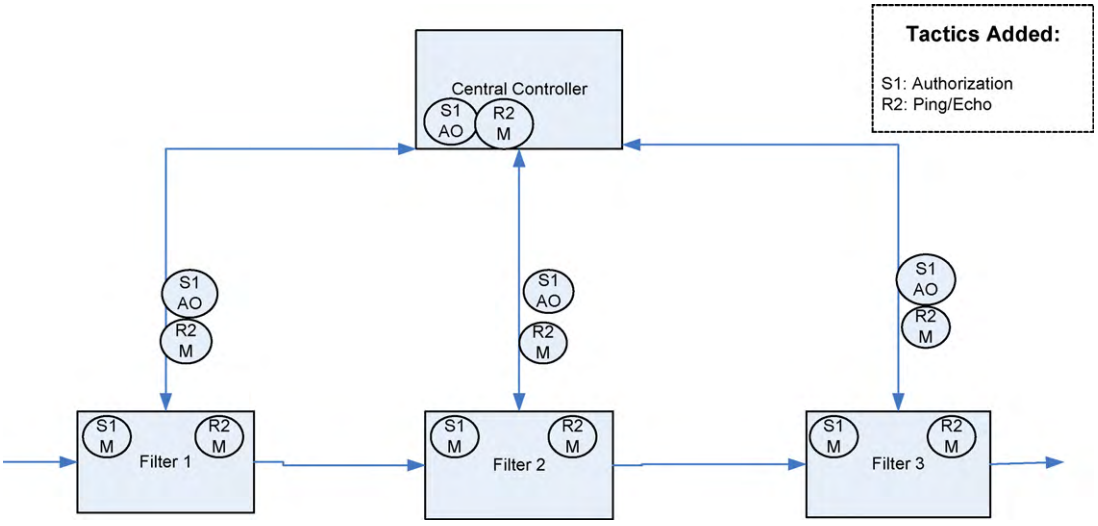

Fig. 8. Addition of Ping/Echo to Pipes and filters with Authorization. Note that we reuse the connectors between the filters and the central controller.

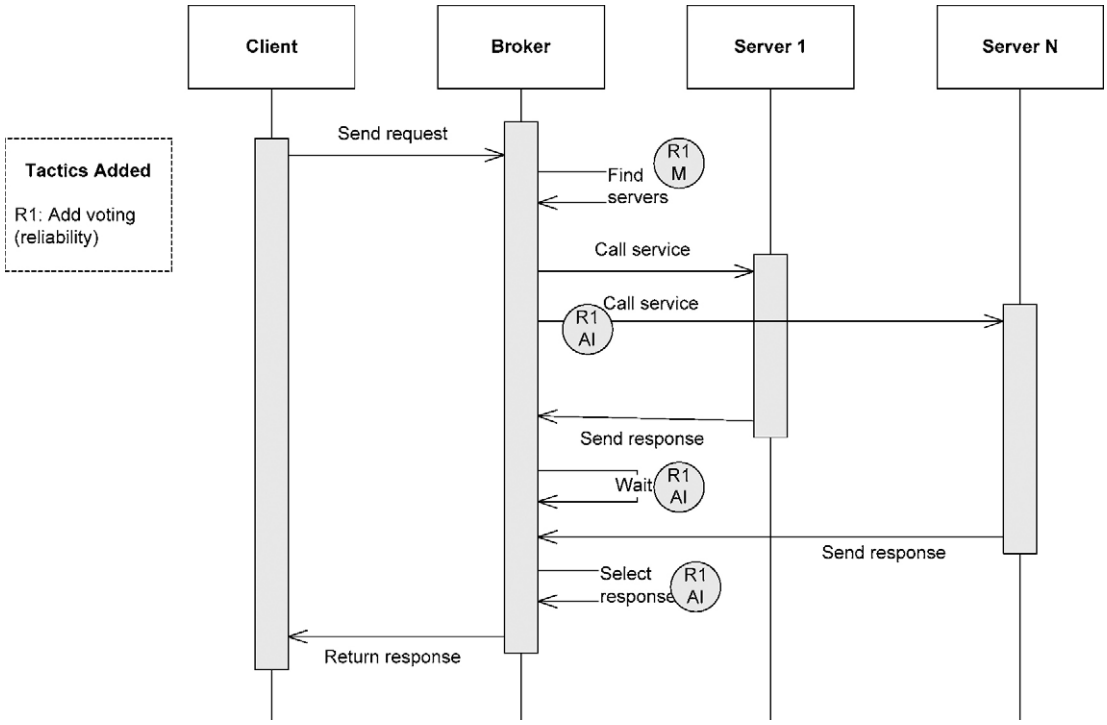

Fig. 9. Broker message sequence diagram with voting added.

through the archive and recover components.) This diagram is problematic because it is not clear what the large rectangles mean. Most appear to designate association, such as the lower rectangle labeled "Operator Windows," but the one we are concerned with, the Statistics rectangle, may in fact designate containment within a common database component. Furthermore, we were unable to obtain the book that contains this diagram, so further information is unavailable. Even without this information, we understand that we must duplicate the data and manage the redundant databases. Clearly, the clients of the data should not have to know about the duplicated database, so that management must happen inside the Statistics rectangle. It appears that if it was not a containment relation before, it certainly is now. We explain the tactic structure in light of the uncertainty of the diagram; for illustration purposes, we have put it in the tactic list. The notation does not specify where it should go.

5.1.2.3. Speech recognition. The following example is of a speech recognition system, based on the Blackboard architecture pattern, from (Shaw, 1991). The blackboard pattern consists of a central data store, called the Blackboard component, a controlling component, and several knowledge sources that are specialized for solving a particular part of the overall task.

We wish to add the reliability tactic Voting. The idea here is that speech recognition can be very difficult, and different algorithms may have different degrees of success at different times. To maximize the chances of accuracy, we implement two or more different algorithms and select the best result among them. (The best result might be obtained by taking the two outputs that are most similar, for example.)

The diagram shows several components within the Blackboard component called "levels." It is not clear whether they form a Layered architecture within the Blackboard, but it does not matter: we want to re-implement the Blackboard component as a whole. We put the tactic circle next to the Blackboard label to show this.

A voting scheme requires a Voter component to make the selection among the results. This appears to be a natural task for the Blackboard Monitor component. We see all these changes reflected in Fig. 12.

5.1.3. Uses for the notation

Clements et al. (2002b) note that two major purposes of architecture documentation are for education and for communication among stakeholders. Our studies given below indicate that it can be useful as a communication tool to show where tactics are to be implemented. It may also be useful in education (e.g. when new members join the team), as it shows quickly and graphically where changes to the architecture have been (or will be) made to implement tactics. In the future, we intend to study and potentially provide evidence about the educational benefits of the annotation.

A potential limitation of the utility of the notation is the complexity of systems; both the number of patterns employed and the number of tactics used. Our studies of software architectures (Harrison and Avgeriou, 2008a), show that most (31 out of 47) architectures use more than one pattern, but only 6 used more than 3 patterns. As shown in the examples and case studies, the notation is adequate for systems with 3 patterns. Of course, the number of tactics is more critical, and some systems can employ numerous tactics (Ozakaya et al., 2008). We see in the previous figures that some tactics affect only one or two components and connectors, while other tactics have much wider impact. If many tactics with wide impact are used, the resulting architecture diagram will be very cluttered, and likely unwieldy. In such a case, one may wish to show either only the latest tactics added, or only the tactic relationships that are particularly notable. This problem is solved if the notation is supported by a tool: a designer can select to show or hide tactics at will.

A potential use is to facilitate identification of areas of compatibility or conflict among the tactics themselves. We have seen where authorization enabled Ping/Echo, and the close relationship of the tactics' implementations is reflected in Fig. 8.

This annotation is also potentially useful in software architecture reviews. Several researchers have documented the worth of software architecture reviews, but have also noted their high cost

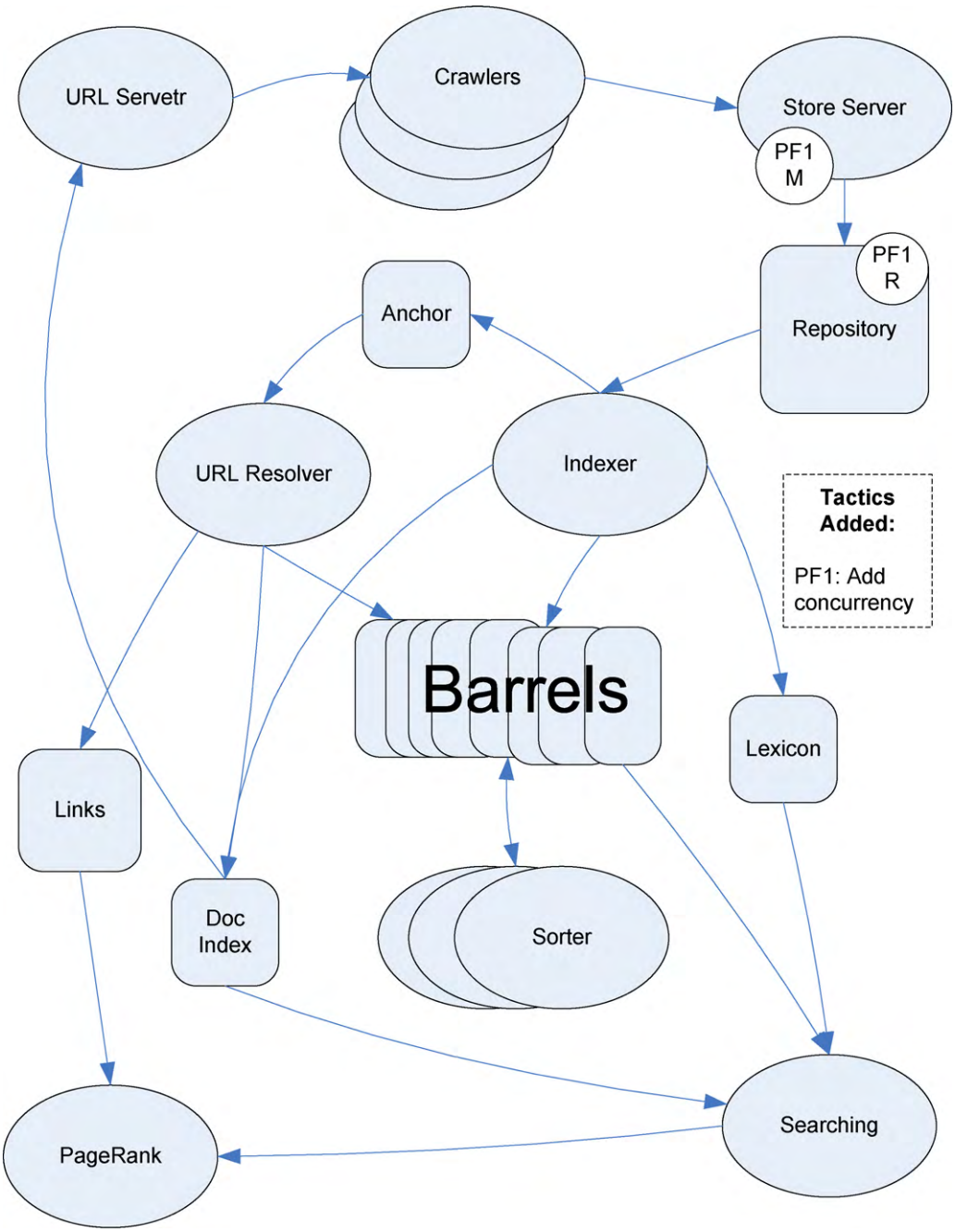

Fig. 10. Google architecture with "add concurrency" tactic.

(Bass et al., 2006a; Abowd et al., 1997; Maranzano et al., 2005; Clements et al., 2002c). We are exploring low-cost architecture reviews, and have annotated architecture diagrams during some reviews as an aid in discussing changes to the architecture. While the results are preliminary, the use of this annotation during architecture reviews appears to be very beneficial. The next section presents three of these case studies.

6. Case studies

We have validated the model and annotation further by annotating diagrams associated with architecture reviews. In some cases, the annotations were added after the review, and annotated architecture diagrams were given to the architects. In other cases, the architecture diagrams were annotated during the review, and the participants were able to easily see where to add the tactics. We describe three of the studies below.

6.1. Case study 1

6.1.1. Product description

We visited company "A" and studied the architecture of its flagship product. (Proprietary concerns prevent giving the real name of the company or revealing details of the product and its architecture.) Company A produces systems that perform sequential processing of certain materials. The systems consist of modules that perform specialized processing; these modules may be combined in various ways to meet the customers' needs. Each module con-

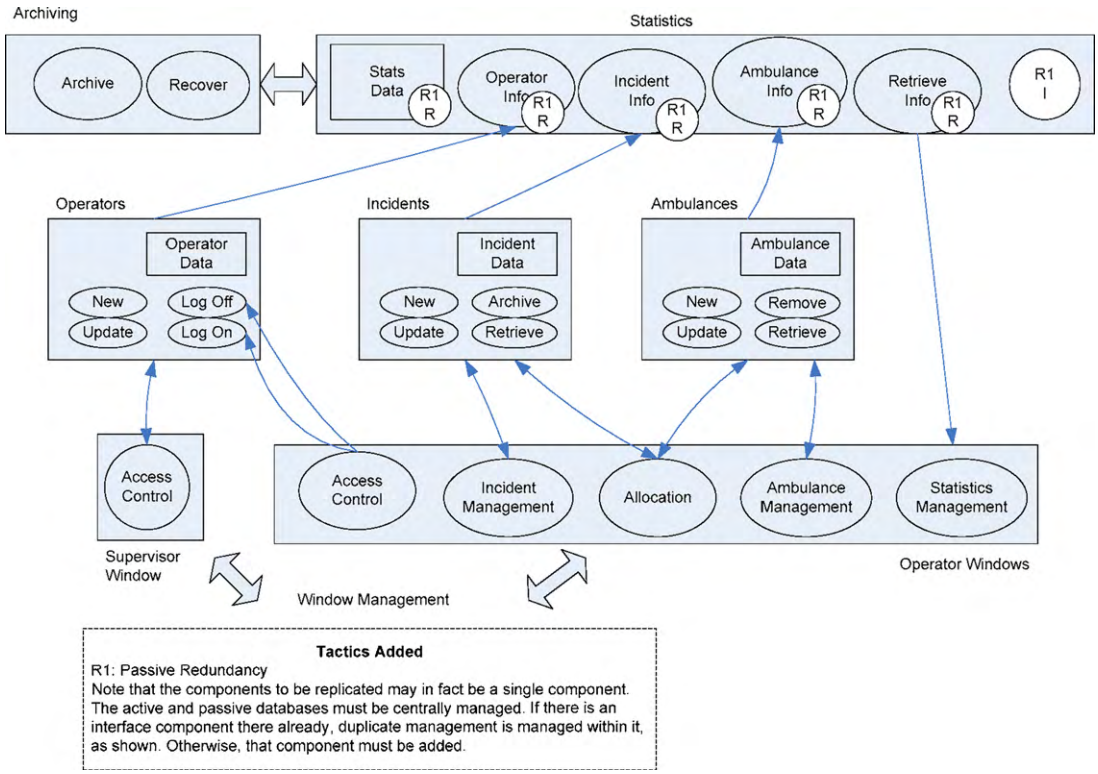

Fig. 11. Ambulance with "passive redundancy" tactic.

sists of specialized hardware and controlling software. Although modules perform their tasks autonomously, they are coordinated by a central controlling module.

The product has several important quality attributes, and particular challenges associated with them. The three key quality attributes are performance, modifiability, and reliability. All three are critical, and it would be difficult to say which is most important.

Performance is a key attribute in that the system must process customers' materials as fast as possible – speed of processing contributes directly to the advertised capacity of the system.

In this case, modifiability means that the system may be configured in many ways depending on the customer's needs.

Reliability consists of processing the material correctly. This is very important in this product; errors cannot be tolerated. However, handling the material is inherently error-prone. Therefore, it is critical that the system have robust fault tolerance mechanisms. It is largely the responsibility of the software to implement fault tolerance.

6.1.2. Architecture

The physical nature of the task, coupled with the need to have flexibility in configuration and high speed processing, led the architects to a Pipes and Filters architecture. The filters consist of hardware components with their associated software. The components operate on physical materials (rather than data, as traditionally found in Pipes and Filters architectures), and pass it to the next filter in the chain. The Pipes are simply physical transfer mechanisms carrying the materials from one to component to another. Fig. 13 shows the basic architecture, without tactic information.

6.1.3. Tactics employed and their impact on the architecture

The Pipes and Filters pattern is a good fit for performance and modifiability (Harrison and Avgeriou, 2008a). However, it is generally not a good fit for reliability (Harrison and Avgeriou, 2008b). We found that measures to improve reliability had significant impact on the architecture.

The most significant reliability tactic employed was a form of transaction rollback to handle faults encountered during materials processing. This required the addition of a central managing component that communicates with each of the filter components. This is adding a component outside of the architecture, resulting in significant change to the architecture. New communication links must be added, and the filter components must add corresponding software.

The second reliability tactic to be implemented is the Ping–Echo tactic. This tactic addresses the potential problem of handling a filter component that crashes or otherwise becomes unavailable. Fortunately, the central manager and the associated communication links added for transaction rollback can be used to implement this tactic. Therefore, implementation of Ping–Echo is accomplished by modifying existing components. But on the filter side, the modifications are potentially significant: Ping–Echo requires that the ping-ed component respond in a timely manner. This requires that each filter implement a prioritized message handling and response scheme. This could perturb normal timely processing of materials, and must be implemented carefully.

Fig. 14 shows the architecture, with annotation of the two tactics described above.

6.1.4. Notes on the architecture

There are two architectural considerations worthy of note. The first is that the central manager can provide other capabilities besides implementing the two tactics given here. The central manager can manage the startup and shutdown of the entire assembly line. It can also dynamically reconfigure an individual component, if necessary. It is likely that even without these two tactics, there would be a need for the central manager component, even though it significantly changes the Pipes and Filters pattern.

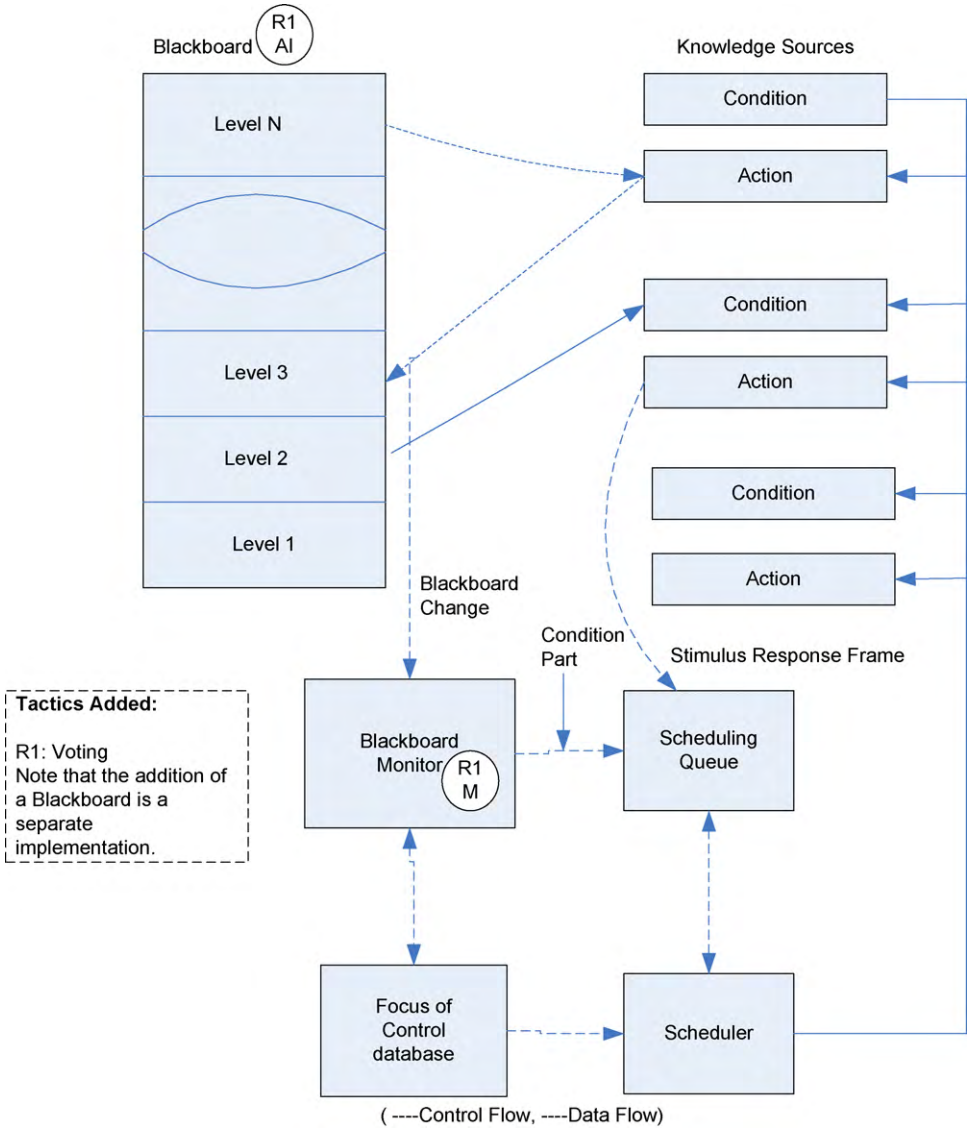

Fig. 12. Speech recognition with "voting" tactic.

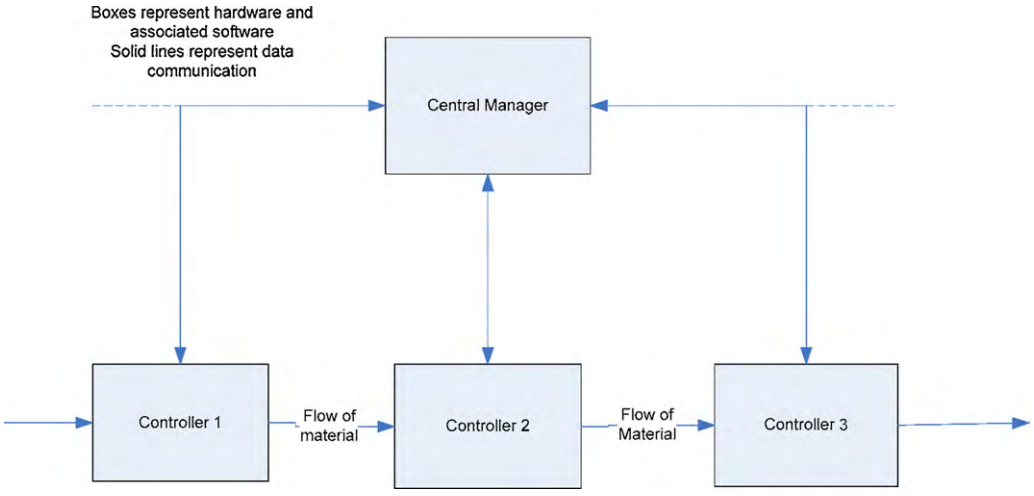

Fig. 13. Company A product architecture (before application of tactics).

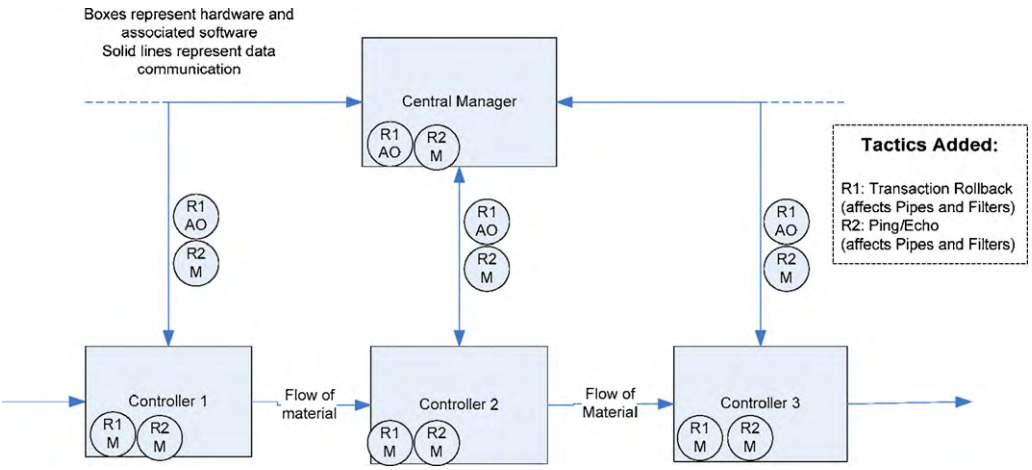

Fig. 14. Company A: product architecture with tactic application shown by annotations.

Second, the failure of a single filter component has ramifications beyond itself. In particular, if a filter component encounters an error or fails completely, the filters that precede it must not continue sending material to it. This requires that the central manager notify controllers to stop processing material. Fortunately, the communication infrastructure is available as part of the transaction rollback tactic, so no components need be added. Details of the messages used for these purposes could be annotated in message sequence diagrams, if desired.

Note that the order of implementing the tactics appears to be irrelevant. While the Ping/Echo tactic uses the infrastructure introduced by Transaction Rollback, if Ping/Echo is implemented first, Transaction Rollback can use the infrastructure introduced by Ping/Echo.

A notable aspect of this architecture is that the tactics of Transaction Rollback and Ping/Echo were part of the initial design of the system; that is, the tactics shaped the architecture from the beginning. The tactic annotation helps illustrate the purpose of the main controller component, and can help show why this architecture deviates from the standard Pipes and Filters architecture.

6.2. Case study 2

6.2.1. Product description

This product is a web-based time tracking system for use by contractors to record time worked on various contracts. It allows a contractor to log in and specify the contract being worked on, and the system records the time spent. It allows the data to be summarized for the purposes of generating payments to contractors.

Because the contractors work from various locations, the system must allow remote access. However, this opens up opportunities for errors, including human errors. A key feature of the system is its handling of erroneous situations. These situations may require the ability to manually correct or update the time data.

The main quality attribute of the system is reliability. Performance (or capacity) is also worthy of note. The following aspects of quality attributes were considered:

1. Reliability (sub-area: availability): It is desirable that the system be available at all times. However, this does not appear to be a critical need: contractors simply call and report when the system is unavailable.
2. Reliability (sub-area: fault tolerance): Faults in the data can be tolerated to a certain extent in that they can be manually corrected. However, this is an area of concern, because it opens many opportunities for error and even fraud. Here are some fault tolerance scenarios that should be handled gracefully:
   a. The worker punches in and neglects to punch out.
   b. The client computer goes down after the worker punches in.
   c. The link between the server and the client server goes down while the worker has punched in.
   d. The worker punches out without punching in.
   e. The worker punches in from two different computers at the same time.
   f. The server goes down.
3. Reliability (accuracy): like fault tolerance, the ability to manually correct entries allows some inaccuracies to be tolerated, but they should be minimized.
4. Performance (capacity): The system should support at least 50 simultaneous users, although this requirement is somewhat flexible. There may be a concern with the performance of the software on the server side, but that is probably a later concern.
5. Performance (response time): Users should see reasonable response (exact response time figures were not given.)

6.2.2. Architecture

The architecture is dominated by the fact that the system is data-centric, and that it must allow remote use. This led to an architecture based on the Repository and Broker patterns. Remote systems employ a thin client that communicates to a central server. Processing of data on the server also led to Layers for part of the system (Fig. 15).

This Diagram shows three prominent patterns. The client communication is handled by the component called “Common”, which is functioning as a Broker. A database (the Repository pattern) underlies the entire system. On the right, we see the main processing of the system contained in three layers. We note that the “Common” component communicates directly with the layers, thus changing or “breaking” the Layers pattern; this was done to improve response time. The change to the Layers pattern is not noted in the architecture diagram, and may be missed by future architects and developers. The next diagram will annotate that modification.

6.2.3. Tactics employed and their impact on the architecture

In order to improve the fault tolerance of the distributed system, the Heartbeat tactic was suggested. The thin client must send regular heartbeats to the server. Heartbeat can be implemented entirely within the Broker structures (consisting of the Broker and the thin clients) with no structural changes. In fact, heartbeat mes-

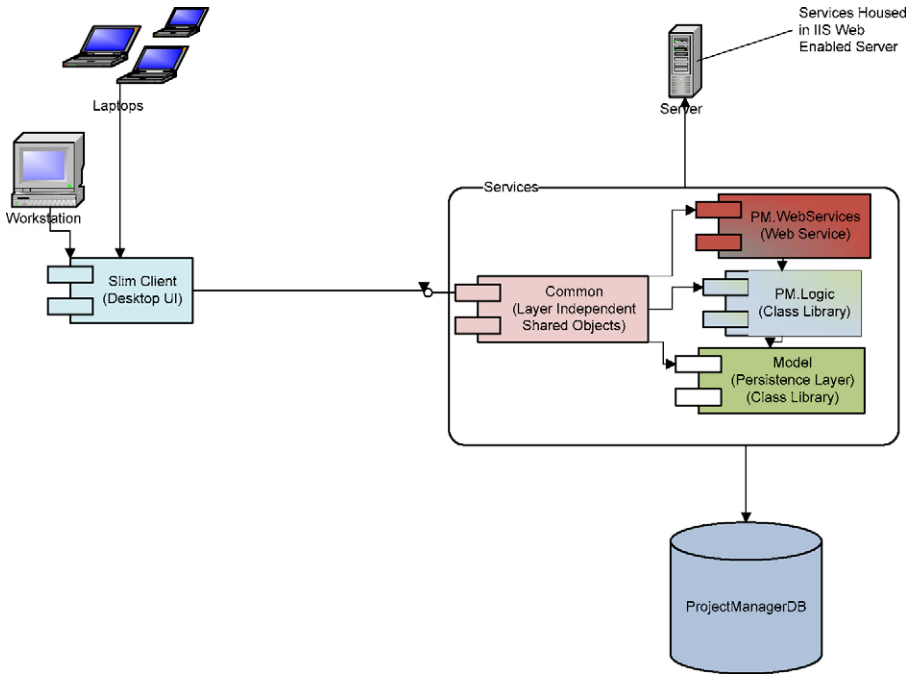

Fig. 15. Time manager architecture (before annotation).

sages often piggyback on other messages; this can be done in this case as well.

In order to prevent data from being corrupted, transaction rollback can be introduced. This tactic often comes as part of commercial databases, so it is implemented within the Repository pattern, with no structural changes needed.

In order to improve performance, the Broker might bypass some of the layers. This involves adding communication links outside the Layers pattern.

Fig. 16 shows the architecture, with the addition of the two tactics described above.

6.2.4. Notes on the architecture

The architects chose to use a different type of architecture diagram than in other example in this text. Yet we see that the same annotation method can be used on different types of diagrams.

The architecture shows the addition of a component used to bypass layers to improve performance. In this case, the common component could easily be used to implement another tactic, namely the Heartbeat tactic. However, the next case study shows where the addition of a component (again, bypassing layers to improve performance) causes difficulty when adding another tactic.

6.3. Case study 3

6.3.1. Product description

This product is a system that supports a magazine subscription clearinghouse business. The system's purpose is to collect, store, and track information about customers and their magazine subscriptions.

Currently, customers mail orders for magazine subscriptions to the company. Clerks then enter the orders into the system. The clerks may work from home, so remote access is required.

Future enhancements may include allowing web-based ordering. At this time it is not needed, because the targeted customers are prison inmates, who have limited web access.

The following quality attributes are important to this system:

1. Reliability (accuracy and fault tolerance): The consequences of inaccurate data are serious: Customers get angry because they do not get the magazines they paid for. And it costs time and trouble to hunt down the error, even to the point of finding the original paper order and verifying it.
2. Performance (capacity): The amount of data being handled is quite large for a system of this size. There are currently around 90 thousand customers, and roughly one million orders. There are around 1000 different magazine titles. However, because each magazine can come from multiple suppliers, the real number of magazines is three to four times that number. The system must handle not only this load, but larger amounts of data as the business expands.
3. Extensibility: There are numerous important capabilities that will be needed in the future, so the system must be easily extended.
4. Security (authorization): It is important to keep the system safe from unauthorized use. In particular, data entry employees may have different privileges.
5. Performance: The system has no hard or soft real time needs, but response time should be quick.

6.3.2. Architecture

The quality attributes played a significant role in the shaping of the architecture. The architects selected a commercial database that provided good reliability (such as transaction rollback.) It also supported the high capacity needed for now and the future. The architects used the Model View Controller (MVC) pattern, which supports extensibility.

The controller part of the system was built using a layered framework (a commercial framework); however, it was found to have slow response time on occasion, so the architect took advantage of another commercial product to bypass layers. This is a common way of "breaking" the Layers pattern in order to improve performance. At this point in the architecture, security needs have not yet been considered. Fig. 17 shows the architecture up to this point.

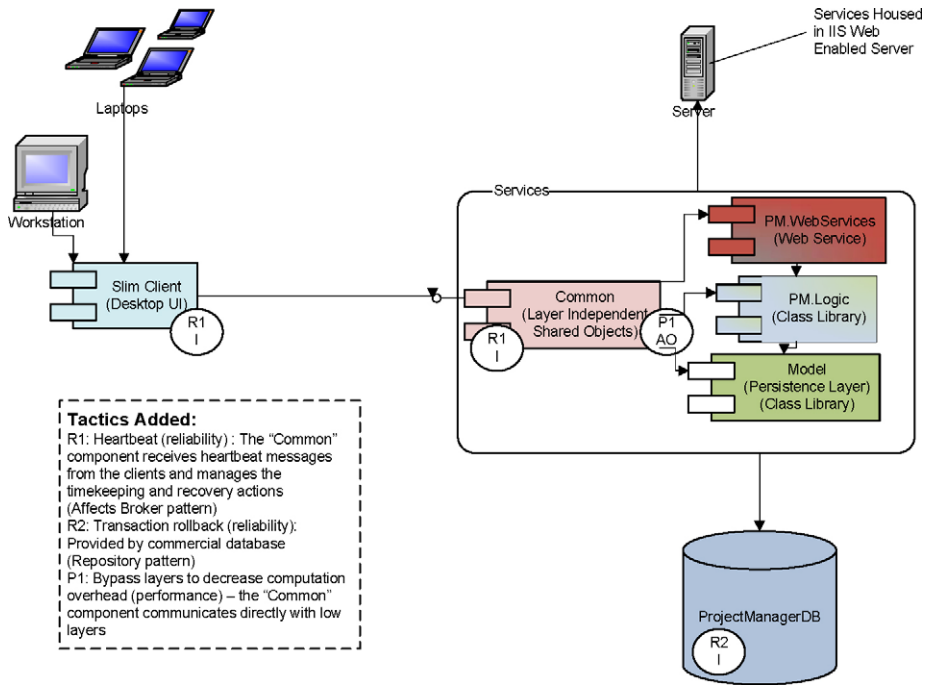

Fig. 16. Time manager product architecture with tactic annotations.

In this diagram, we can see the Repository, as well as the Model, View, and Controller components. We note that the views are remote, comprising the client side of the client–server pattern. We also see the component that bypasses the layers. It is reasonably clear without additional annotation.

However, authorization had not yet been added. In a layered architecture, authorization can be easily added as an initial layer – adding a component within the pattern structure (minor impact.) In this case, though, there were two paths through the system from the views. Each required authentication. So a layer was added to the Controller layers (added within the pattern), and another component was added before the component that bypasses the layers. This new component is outside any of the patterns. Of particular concern is that two different components perform the same function, leading to the probability of duplicated code.

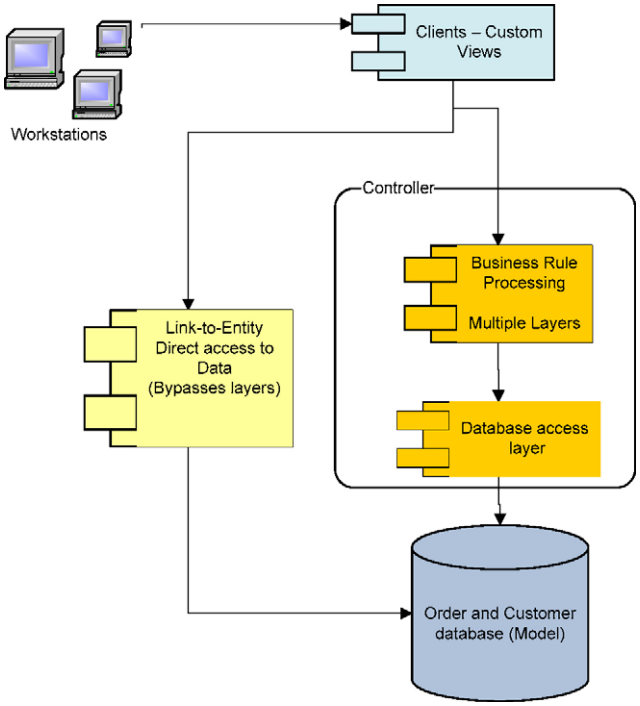

Fig. 17. Subscription management system architecture (before annotation).

6.3.3. Tactics employed and their impact on the architecture

Fig. 18 shows the complete architecture, after the authorization components were added. We have annotated the diagram with tactic information. The annotation highlights the components that have been added.

6.3.4. Notes on the architecture

This architecture illustrates the case where the addition of components outside the architecture patterns has a snowball effect – adding the layer bypass component caused an extra authorization component to be added, with duplicated code (Note that adding a single authorization component above both components was not feasible – it ended up defeating the performance gains that the direct access component provided.) This has the potential to adversely impact system maintenance. It is not clear whether it will have any impact on extensibility, an important quality attribute for this system. However, the close relationship between maintainability and extensibility makes this architecture a cause for concern.

The architecture diagram, together with the annotation, should encourage consideration of alternative architectures. The following three alternatives might be considered:

1. Could the authorization be handled in the views component? This is unlikely, since that would mean that the authorization happens on a remote client, which itself is a security vulnerability.
2. Could the authorization layer handle authorization for both the layers and the bypassing component? This appears to be a reasonable solution, but has one notable weakness. The authorization component would have to make the decision about whether to route requests through the layers or through the

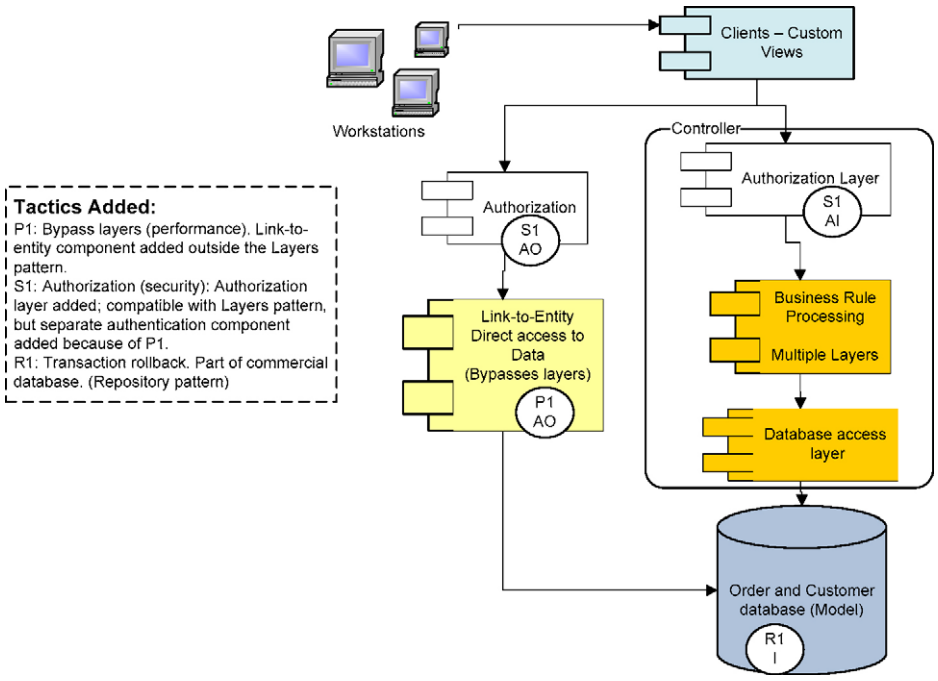

Fig. 18. Subscription management system architecture with tactic annotations.

bypass. This would require the authorization component to have detailed application knowledge, which is well beyond its job.
3. Could the bypass be removed altogether? This is a tradeoff decision between performance and maintainability. The architects decided that the benefits of improved performance outweighed the liabilities of increased maintenance difficulty.

6.4. Lessons learned

These case studies show several benefits of annotating architecture diagrams with tactic information in this manner. We derived the participant observations through informal interviews with participants in two architecture reviews, where the architecture diagrams were annotated during the review. In addition, three architecture diagrams were annotated and emailed to the architects. In both cases, participants were asked whether they observed benefits, and if so, what. We noted the following benefits:

1. The participants observed that it clearly shows which components must be modified, and gives a high-level idea of the nature of the changes that must be made or have already been made.
2. The participants observed that it shows where tactics interact with each other. For example, the third case study shows the addition of tactics for both performance and security, where an additional authorization component was required because of the component added for performance.
3. We observed that the annotation can be used both to show the addition of tactics to existing architectures and to show tactics in the original architecture.
4. We observed that the annotation is adaptable to various styles of component and connector architecture documentation. It can annotate both the components and the connectors with the same notation. We established this by annotating different documentation styles, as shown in the examples.
5. We observed that the annotation is easy to add; we were able to annotate diagrams during the reviews. This is important, as it has been noted that effort required can hinder the amount of architecture documentation done (Jansen and Bosch, 2004).
6. We observed that the amount of documentation needed is potentially small. Where documentation of interaction of architecture patterns and tactics exists, it can be leveraged to minimize the amount of writing needed. Additional insights can be added to the tactic description list as needed to increase clarity.
7. We observed that for simple architecture diagrams, the notation is small enough that the architecture itself is not obscured. In the diagrams above, the size of the boxes representing components were rather small, but the tactic circles could still be placed so that the architecture itself was clear.

We also observed several limitations to the annotation. They include the following:

1. The notation gives information about the location and general type of changes needed to implement tactics. However, it does not necessarily give specific information about the changes. Of course, there is a tradeoff between information given and both effort and clarity of architecture diagrams. This approach attempts to maximize clarity and reduce effort; however additional documentation might be desirable, especially for large projects.
2. The notation is most effective when it can refer to detailed information about the interaction of patterns with tactics. Unfortunately, much of that information in not published. This paper shows some samples of such data, but remaining information must be written.
3. The consideration of design time tactics was out of the scope of our work. Investigation of applying the notation to design time tactics is possible future work.

7. Related work

Bass et al. (2003) have described many important tactics, and note that patterns and tactics interact. This paper explores this relationship in depth.

Many architecture pattern descriptions include some explanation or a pattern's compatibility or incompatibility with certain quality attributes (Maranzano et al., 2005). The interaction has been described across more patterns and quality attributes in (Harrison and Avgeriou, 2007a), though at a very high level. The interaction is characterized in more detail in (Harrison and Avgeriou, 2008a). It has been explored in considerable detail for the well-known tactics of reliability interacting with one or two architecture patterns (Harrison and Avgeriou, 2008b). This paper describes the types of interactions and presents a method of documenting the interactions.

Numerous methods of architecture design have been proposed; several important such methods have been generalized by Hofmeister et al. (2005). The general model consists of three activities: Architectural analysis identifies the architecturally significant requirements (ASRs) of the system; namely those requirements that influence its architecture. The second activity is architecture synthesis, the creation of candidate architecture solutions to address the ASRs. Architecture evaluation, the third activity validates the candidate solutions against the ASRs. Our model shows how quality attributes are architecturally significant, and provides support for determining tradeoffs during synthesis and evaluation. The model explicitly supports making such decisions in pattern-based architecture design (PDAP) (Harrison and Avgeriou, 2007b), and in quality attribute-based reasoning frameworks as proposed by Bachmann et al. (2005). The annotation method is also compatible, and can be used in architecture synthesis and evaluation.

Architecture reviews have been identified as an important way to improve the quality of the software architecture (IEEE, 2000; Maranzano et al., 2005; Bianco et al., 2007; Obbink et al., 2002). We have found that annotating an architecture diagram can be a useful activity during an architecture review.

In ATAM and ADD (Bass et al., 2003), the analysis of architectural tradeoffs is an important part of the architectural decision making process. In particular, one considers tradeoffs among tactics. Our analysis of the nature of impact of tactics on architecture patterns supplements tradeoff analysis: it helps architects understand and visualize where the impact of a tactic occurs, and how great this impact is likely to be. This information helps provide specific substance to be used in tradeoff analysis.

Several of these architecture methods center on various views of the system. Clements et al. (2002a) describe general viewtypes of module, component and connector, and allocation. The Siemens' 4 Views method (Hofmeister et al., 2005) has four views: conceptual, execution, module and code architecture. The Rational Unified Process 4 + 1 Views (Kruchten, 1995) centers on four views to describe the architecture: logical view, process view, implementation view, and deployment view. As shown, our annotation method is quite flexible, but is most naturally applied to Siemens' code architecture view and to the RUP logical view. It fits well within Clements' component and connector viewtype.

Numerous formal models and architecture description languages (ADLs) have been developed for expressing architecture (Allen and Garlan, 1994; Allen, 1997; Shaw et al., 1995). They have different levels of support for modeling and representing patterns; however the most popular ADLS have components and connectors as central building blocks of architecture descriptions. As we have shown, our annotations can be applied to components and connectors in architecture diagrams. We have not proposed any extensions to text-based ADLs that provide the annotation information; however the common focus on components and connectors makes it clear that such augmentations are easily made.

Zdun (Zdun, 2007) has proposed design space analysis as a method for the selection of architecture patterns and pattern variants. Quality attribute information is a key part of the design space information. Changes to the patterns to accommodate tactic implementation might result in what Zdun calls "pattern variants." Thus Zdun's work appears to be highly compatible with this work.

The IEEE 1471 standard (IEEE, 2000) describes a meta-model of software architecture that includes, among other things, multiple views of the architecture as well as the satisfaction of stakeholder concerns. Our work focuses on stakeholder concerns of quality attributes, and shows how views of the architecture (particularly component and connector-based views) can be annotated with such information. Thus it is compatible with this meta-model.

An important area of research is the interaction of multiple tactics. Nearly every non-trivial system implements multiple tactics, and they might not be independent of each other. We have seen in Fig. 8 how a tactic can be implemented to take advantage of a previously implemented tactic. Thus it appears that the order of implementing tactics is significant. The Architecture-Driven Design (ADD) method (Bass et al., 2003) acknowledges this. Bass et al. build on this to show how patterns and tactics show rationale and design changes over time (Bass et al., 2006b). This work captures ordering of tactic implementation. One could also capture a history of architecture snapshots, showing the addition of snapshots over time.

8. Future work

There are several areas that require further research. These include the following.

Our studies have shown that architects consider the annotation of architecture diagrams with tactic implementation information useful. But longer term, how is it used? How useful is it for maintainers who are learning the system architecture? This is an area for long-term study.

We have used the interaction information to help identify issues during architecture reviews (as noted in the case studies), and have begun using the annotations in the reviews as well. We expect to use them further in order to better understand their benefits as well as limitations in architecture reviews. It may also be beneficial to tie the tactics' implementations to the risk themes of architecture reviews as described by Bass et al. (2006a).

The interaction of multiple tactics as well as tactics on multiple patterns should have additional study. We are considering how the magnitude of the impact of tactics changes when considered in the context of multiple patterns.

Another issue of multiple tactics is how the annotation method scales. At what point does the annotation become less useful because it clutters the architecture diagram? While we have found it scales up through six tactics, we do not know how it scales to very large systems.

We have concentrated on the run time tactics as described in (Bass et al., 2003). We find that the other tactics they describe, design time tactics, tend to cut across all design partitions, and are implemented implicitly in the code. Therefore, they are not as good a fit for the model as are the run time tactics, nor do they have as well-defined effects on patterns. The model and the design time tactics should be studied in more detail to determine how the design time tactics can be represented in the model and annotation. Besides refinement of the model and annotation, it will lead to greater insights about the nature of design time tactics versus run time tactics.

9. Conclusions

It has long been known that architecture patterns and quality attributes are not independent, but have significant interaction with each other. We have found that this relationship is very rich,

and involves the implementation of quality attributes through tactics.

The interaction among patterns and (runtime) tactics falls into several general categories, based on the type of changes needed to the pattern in order to implement the tactic. The amount of work required to implement a tactic in an architecture that uses a pattern depends on the type of change, as well as the volume of change needed. However, details of these interactions are different for each pattern/tactic pair, and must be investigated and documented.

The interaction of tactics and patterns can easily be shown in architecture diagrams using a simple method of annotation. This annotation method is adaptable to many different types of architecture diagrams and documentation. We have shown its use in various types of structural and behavioral architecture diagrams. We have used it to annotate several real architecture diagrams, both during and after architecture reviews. It has shown itself to be useful for illustrating the changes required.

Appendix A.

Tables A1 and A2 are short descriptions of some patterns and tactics used in this paper.

Table A1
Common architecture patterns (From Avgeriou and Zdun, 2005).

Pattern	Description
Client–server	There are two kinds of components, clients and servers. The client requests information or services from a server. Clients are optimized for their application task, whereas servers are optimized for serving multiple clients.
Broker	The broker hides and mediates all communication between clients and servers in a system.
Layers	A system is structured into layers so that each layer provides a set of services to the layer above and uses the services of the layer below.
Pipes and filters	A complex task is divided into several sequential sub-tasks, each of which is implemented by a separate independent component, a filter, which handles only this task.
Blackboard	A complex task is divided into smaller sub-tasks for which deterministic solutions are known. The Blackboard uses the results of its clients for heuristic computation and step-wise improvement of the solution. A control component monitors the blackboard and coordinates the clients.

Table A2
Common tactics (from Bass et al., 2003).

Tactic (associated QA)	Description
Ping–Echo (reliability)	One component issues a ping and expects to receive back an echo, within a predefined time, from the component under scrutiny.
Introduce concurrency (performance)	Process requests in parallel by creating additional threads to process different sets of activities.
Authenticate users (security)	Ensure that a user or remote computer is actually who it purports to be through passwords, etc.
Checkpoint/rollback (reliability)	If a system fails with a detectably inconsistent state, restore it using a previous checkpoint of a consistent state.

References

Abowd, G., et al., 1997. Recommended Best industrial Practice for Software Architecture Evaluation. Software Engineering Institute, Technical Report CMU/SEI-96-TR-025.

Allen, R.J., 1997. A formal approach to software architecture. Doctoral Thesis, UMI Order Number: AAI9813815. Carnegie Mellon University.

Allen, R., Garlan, D., 1994. Formalizing architectural connection. In: Proc. International Conference on Software Engineering (ICSE), Sorrento, Italy, May 16–21. IEEE Computer Society Press, pp. 71–80.

Avgeriou, P., Zdun, U., 2005. Architectural patterns revisited – a pattern language. In: 10th European Conference on Pattern Languages of Programs (EuroPLoP).

Bachmann, F., et al., 2005. Designing software architectures to achieve quality attribute requirements. IEE Proceedings 152 (4), 153–165.

Bachmann, F., Bass, L., Nord, R., 2007. Modifiability Tactics. Software Engineering Institute, Technical Report CMU/SEI-2007-TR-002.

Barbacci, M.R., et al., 2003. Quality Attribute Workshops (QAWs), 3rd edn. Software Engineering Institute, Technical Report CMU/SEI-2003-TR-016.

Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice. Addison-Wesley.

Bass, L., et al., 2006a. Risk Themes Discovered Through Architecture Evaluations. Software Engineering Institute, Technical Report CMU/SEI-2006-TR-012.

Bass, L., Clements, P., Nord, R., Stafford, J., 2006b. Capturing and using rationale for a software architecture (chapter). In: Dutoit, A., McCall, R., Mistrick, I., Paech, B. (Eds.), Rationale Management in Software Engineering. Springer-Verlag.

Bianco, P., Kogtermanski, R.L., Merson, P., 2007. Evaluating a Service-Oriented Architecture. Software Engineering Institute, Technical Report CMU/SEI-2007-TR-015.

Booch, G., 2007. Handbook of Software Architecture: Gallery, http://www.booch.com/architecture/architecture.jsp?part=Gallery (accessed 10.09.07).

Bosch, J., 2000. The Design and Use of Software Architectures. Addison-Wesley, London.

Brin, S., Page, L., 1998. The anatomy of a large-scale hypertextual web search engine. In: W3C International World Wide Web Conference, p. 11.

Buschmann, F., et al., 1996. Pattern-Oriented Software Architecture: A System of Patterns. Wiley.

Clements, P., et al., 2002a. Documenting Software Architectures: Views and Beyond. Addison-Wesley.

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

Clements, P., Kazman, R., Klein, M., 2002c. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley.

Cloutier, R., 2006. Applicability of patterns to architecting complex systems. Doctoral Dissertation, Stevens Institute of Technology.

Fernandez, E.B., Ortega-Arjona, J.L., 2009. The Secure Pipes and Filters Pattern. In Proceedings of the 2009 20th international Workshop on Database and Expert Systems Application (August 31 - September 04, 2009). DEXA. IEEE Computer Society, Washington, DC, 181-185. doi:http://dx.doi.org/10.1109/DEXA.2009.55.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA.

Harrison, N., Avgeriou, P., 2007a. Leveraging architecture patterns to satisfy quality attributes. In: First European Conference on Software Architecture, Madrid, September 24–26, 2007. Springer, LNCS.

Harrison, N., Avgeriou, P., 2007b. Pattern-driven architectural partitioning – balancing functional and non-functional requirements. In: First International Workshop on Software Architecture Research and Practice (SARP'07), San Jose, CA, July 1–6, 2007. IEEE Computer Society Press.

Harrison, N., Avgeriou, P., 2008a. Analysis of architecture pattern usage in legacy system architecture documentation. In: 7th Working IEEE/IFIP Conference on Software Architecture, Vancouver, 18–22 February, pp. 147–156.

Harrison, N., Avgeriou, P., 2008b. Incorporating fault tolerance techniques in software architecture patterns'. In: International Workshop on Software Engineering for Resilient Systems (SERENE'08), Newcastle upon Tyne (UK), 17–19 November, 2008. ACM Press.

Harrison, N., Avgeriou, P., Zdun, U., 2006. Focus group report: capturing architectural knowledge with architectural patterns. In: 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006), Irsee, Germany.

Harrison, N., Avgeriou, P., Zdun, U., 2007. Architecture patterns as mechanisms for capturing architectural decisions. IEEE Software 24 (4).

Hofmeister, C., Kruchten, P., Nord, R.L., Obbink, H., Ran, A., America, P., 2005. Generalizing a model of software architecture design from five industrial approaches. In: 5th Working IEEE/IFIP Conference on Software Architecture (WICSA), November 06–10, 2005. IEEE Computer Society, pp. 77–88.

IEEE Recommended Practice for Architectural Description for Software-Intensive Systems. IEEE Computer Society, October 9, 2000.

International Standards Organization. Information Technology – Software Product Quality – Part 1: Quality Model, ISO/IEC FDIS 9126-1, 2001.

Jansen, A., Bosch, J., 2004. Evaluation of tool support for architectural evaluation. In: 19th IEEE Intl Conference on Automated Software Engineering (ASE2004), Linz, Austria, September 2004, pp. 375–378.

Jansen, A.G., Bosch, J., 2005. Software architecture as a set of architectural design decisions. In: 5th Working IEEE/IFIP Conference on Software Architecture (WICSA), November 06–10, pp. 109–119.

Kruchten, P., 1995. The 4 + 1 view model of architecture. IEEE Software 12 (6).

Maranzano, J., et al., 2005. Architecture reviews: practice and experience. IEEE Software 22 (2), 34–43.
O'Brien, L., Bass, L., Merson, P., 2005. Quality Attributes and Service-Oriented Architectures. Software Engineering Institute, Technical Report CMU/SEI-2005-TN-014.
Obbink, H., Kruchten, P., Kozaczynski, W., Hilliard, R., Ran, A., Postema, H., Lutz, D., Kazman, R., Tracz, W., Kahane, E., 2002. Report on software architecture review and assessment (SARA). In: 24th International Conference on Software Engineering (ICSE), Orlando, Florida, May 19–25. ACM, pp. 675–1675.
Ozakaya, I., Bass, L., Sangwan, R., Nord, R., 2008. Making practical use of quality attribute information. IEEE Software Special Issue on Software Quality Requirements March/April.
Rosik, J., Le Gear, A., Buckley, J., Ali Babar, M., 2008. An industrial case study of architecture conformance. In: Second ACM-IEEE international Symposium on Empirical Software Engineering and Measurement, Kaiserslautern, Germany, October 09–10, 2008. ACM, pp. 80–89.
Schmidt, D., Stal, M., Rohnert, H., Buschmann, F., 2000. Pattern-Oriented Software Architecture: Patterns for Concurrent and Distributed Objects. Wiley.
Shaw, M., 1988. Toward Higher-Level Abstractions for Software Systems, Tercer Simposio Internacional del Conocimiento y su Ingerieria (October) 55–61. Reprinted in Data and Knowledge Engineering, 5 (1990), 19–28.
Shaw, M., 1991. Heterogeneous design idioms for software architecture. In: IEEE Workshop on Software Specification and Design, October, p. 5.
Shaw, M., 1996. Procedure calls are the assembly language of software interconnection: connectors deserve first-class status. In: Lamb, D. (Ed.), Studies of Software Design, Proceedings of a 1993 Workshop. Springer-Verlag, pp. 17–32, Lecture Notes in Computer Science, No. 1078.
Shaw, M., Garlan, D., 1996. Software Architecture: Perspectives on an Emerging Discipline. Addison-Wesley.
Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., Zelesnik, G., 1995. Abstractions for software architecture and tools to support them. IEEE Transactions on Software Engineering 21 (April (4)), 314–335.
Sommerville, I., 2007. Software Engineering, 8th edn. Addison-Wesley Longman.
Stevens, R., Brook, P., Jackson, K., Arnold, S., 1998. Systems Engineering. Prentice Hall, Englewood Cliffs, NJ.
Wood, W.G., 2007. A Practical Example of Applying Attribute-Driven Design (ADD), Version 2.0. Software Engineering Institute, Technical Report CMU/SEI-2007-TR-005.
Zdun, U., 2007. Systematic pattern selection using pattern language grammars and design space analysis. Software: Practice & Experience 37 (9), 983–1016, Wiley.

Neil Harrison is an assistant professor of computer science at Utah Valley University in Orem, Utah. Before that, he was a distinguished member of technical staff at Avaya Labs, where he developed communications software led software development teams. He has studied software development organizations for over ten years and is a co-author of *Organizational Patterns of Agile Software Development*. He has published numerous articles on software patterns, effective organizations, and software testing. He is acknowledged as the world's leading expert on pattern shepherding and the pattern conference (PLoP) shepherding award is named after him. Professor Harrison has a BS in Computer Science with High Honors and University Scholar Designation from Brigham Young University, and an MS in Computer Science from Purdue University. He is a member of ACM, The Hillside Group Board of Directors, and Hillside Europe.

Dr. Paris Avgeriou is Professor of Software Engineering at the Department of Mathematics and Computing Science, University of Groningen, the Netherlands. He heads the Software Engineering research group since September 2006. He has participated in a number of national and European research projects on software engineering, that are directly related to the European industry of Software-intensive systems. He has been co-organizing international workshops in conferences such as ICSE, ECOOP, ICSR, UML, ACM SAC and editing special issues for journals like IEEE Software. He is in the editorial board of Springer TPLOP. He is a member of IEEE, ERCIM, Hillside Europe and acts as a PC member and reviewer for several conferences and journals. He has received awards and distinctions for both teaching and research and has published more than 80 articles in peer-reviewed international journals, conference proceedings and books. His research interests concern the area of software architecture, with a strong emphasis on architecture modeling, knowledge, evolution and patterns.