Implementing Reliability: the Interaction of Requirements, Tactics and Architecture Patterns

Neil B. Harrison [1, 2], Paris Avgeriou [1]

[1] Department of Mathematics and Computing Science, University of Groningen, Groningen, the Netherlands

[2] Department of Computer Science, Utah Valley University, Orem, Utah, USA

harrisne@uvsc.edu, paris@cs.rug.nl

Abstract. An important way that the reliability of a software system is enhanced is through the implementation of specific run-time measures called runtime tactics. Because reliability is a system-wide property, tactic implementations affect the software structure and behavior at the system, or architectural level. For a given architecture, different tactics may be a better or worse fit for the architecture, depending on the requirements and how the architecture patterns used must change to accommodate the tactic: different tactics may be a better or worse fit for the architecture. We found three important factors that influence the implementation of reliability tactics. One is the nature of the tactic, which indicates whether the tactic influences all components of the architecture or just a subset of them. The second is the interaction between architecture patterns and tactics: specific tactics and patterns are inherently compatible or incompatible. The third is the reliability requirements which influence which tactics to use and where they should be implemented. Together, these factors affect how, where, and the difficulty of implementing reliability tactics. This information can be used by architects and developers to help make decisions about which patterns and tactics to use, and can also assist these users in learning what modifications and additions to the patterns are needed.

1 Introduction

Software reliability has been defined in ISO 9126 as "The capability of the software product to maintain a specified level of performance when used under specified conditions." [1]. This standard states three key components of reliability: fault tolerance, recoverability, and maturity, including availability. Fault tolerance is, "The capability of the software product to maintain a specified level of performance in cases of software faults or of infringement of its specified interface." Recoverability is, "The capability of the software product to re-establish a specified level of performance and recover the data directly affected in the case of a failure." Maturity is "The capability of the software product to avoid failure as a result of faults in the software." Availability is "The capability of the software product to be in a

state to perform a required function at a given point in time, under stated conditions of use" [1]. Software that is highly reliable must exhibit all these characteristics.

Designing and implementing highly reliable software is very challenging. Besides the fact that the software to make a system reliable is very exacting, it can affect much of the system, and require a significant amount of software: over half of the millions of lines of code written for the 5ESS® Switching System were devoted to error handling. Fortunately, software designers have come up with numerous measures to improve software reliability, based on extensive experience. These measures are implemented in the software, and are designed to help make the software tolerant to faults. These faults include, but are not limited to hardware failures, errors in data, or bugs in the code itself. Many of these measures are well understood and have been documented. Utas [3] describes many such measures, as does Hanmer [4]; both are based on extensive experience in designing and developing carrier-grade telecommunication systems. Hanmer and Utas refer to these as reliability patterns. Some similar measures have been described by Bass et al [2] and called "tactics." For the sake of clarity and simplicity, we call all these measures tactics. The tactics identified by Bass et al [2] address the aforementioned components of reliability as follows: how to detect faults (addressing fault tolerance and maturity), how to prepare for recovering from faults (addressing availability and recoverability), how to recover after a fault (addressing recoverability), and how to prevent faults from causing failures (addressing fault tolerance and maturity).

However, even with the knowledge of the reliability tactics, one must still design and implement them in the system being developed. The difficulty of doing so depends in part on the nature of the tactic to be implemented. The implementation of some tactics requires some code to be written in nearly every component of the system architecture, while other tactics may be implemented with only limited impact. It depends on the tactic.

A given tactic also has different interactions with different architectural structures. Several architectural structures are commonly used, and are called architecture patterns [5] or architectural styles [6, 7]. The compatibility between several common architecture patterns and several common reliability tactics has been investigated [8]. The information about their compatibility is highly useful, because it may help us avoid tactics (or patterns) that are incompatible with the patterns (or tactics) being used. Of course, the harder it is to implement a tactic, the more error-prone the implementation is likely to be. The compatibility information is so far limited to one-to-one relationships: the compatibility of a single tactic with a single architecture pattern. But nearly all commercial systems are complex: they contain multiple architecture patterns (see [9]), and use multiple reliability tactics (see [10]).

But this is not all: every system is different, and has different constraints. Constraints, such as functional, non-functional, and business requirements, earlier design decisions and physical limitations, also affect the structure and behavior of the system. In this paper, we are particularly interested in requirements related to reliability: they are closely tied to the reliability tactics and the architecture patterns. This leads to the key question for this work:

How do the nature of tactics, software architecture patterns, and requirements interact with each other to influence the achievement of reliability in a software architecture?

We have tried to answer this question through a typical research cycle of grounded theory consisting of the following: we first looked at the tactics themselves, we then investigated the interaction between tactics and the pattern structures and finally we looked into an actual system design which included reliability requirements. We found three general ways that the nature of tactics influences the architecture. We found regular ways that multiple architecture patterns interact with tactics. And we found that requirements affect the tactics in two general ways. To fully understand the tactic impact, selection and implementation, one must consider all these factors.

The main contribution of this work is that it provides information into how these factors influence the implementation of tactics, which is indicative of the effort needed as well as difficulty in implementation and future system maintenance. This information can be used to make architectural tradeoff decisions, as well as in development planning. This knowledge is important when one designs even moderately complex software architectures (two or more patterns) that must be reliable.

In the following sections, we describe how tactics are implemented in complex systems. In section 2, we give background and describe the challenge of implementing tactics in complex systems. Section 3 describes the three factors that influence how and where tactics are implemented. Throughout sections 2 and 3, we use a running example of an airline booking system. In section 4, we describe how the information can be used in a practical setting, and how it fits with typical software architecture processes. We provide a case study and other validation in section 5. Section 6 describes related work, and section 7 describes future work.

2 Background: Architecture Tactics and Patterns

Designing the architecture of a software system consists of designing the structure and behavior of the system. This comprises making decisions about the software elements, the externally visible properties of those elements, and the relationships among them (from [2]). One of the key challenges of software architecture is to make decisions that satisfy not only the functional requirements of the system, but also the non-functional requirements, or quality attributes. One of the most important quality attributes is often reliability. Architectural decisions may support each other, but often conflict each other; thus tradeoffs are a common aspect of architecting. Two of the most important types of decisions are those concerning how to meet quality attributes (architecture tactics), and decisions about the overall structure and behavior (architecture patterns). These are discussed in turn below.

Let us consider a system to book airline tickets, which will be used as a running example. The system has multiple simultaneous users, distributed geographically. Two of the most important requirements related to reliability are as follows:

1. Availability: the system must always be available for use; the consequences are potentially significant loss of revenue. After all, if a customer can't access the reservation system, he or she will turn to a competitor.

2. Data integrity: data must be correctly recorded, and must be accurately recovered as necessary. Transactions must be completed accurately.

(Note that these are not the only reliability requirements on such a system, but are the two we will consider in this example.) A tactic is a design decision that is intended to improve one specific design concern of a quality attribute. The tactics concerning reliability are especially important. For example, a reliability design concern is how to detect whether a component is unable to perform its function, so that it can be restarted. One tactic to implement this design concern is "Heartbeat": each component must send periodic heartbeat messages to a central controller (or, alternatively, to other components.) If a heartbeat message is not received from a component after a specified period, the component is assumed to be no longer sane and must be restarted.

Many tactics have been identified [2], including several important reliability tactics. Other tactics have also been identified, although they might not specifically be referred to as tactics. (See [3] and [4], for example).

Some tactics are related to each other in that they improve the same reliability concern; Bass et al refer to these as "design concerns." In some cases, such tactics are alternatives to each other. For example, detecting faulty processing is a design concern. The tactics to address this design concern are Ping-Echo, Heartbeat, and Exceptions. Ping-Echo and Heartbeat are alternatives to each other.

In the airline reservation system, we analyze different types of faults that the system may experience. These include:

- Bugs in software, including infinite loops, deadlock, and livelock, can cause software components to hang. Such problems can make the system unavailable.
- Hardware failures or software bugs can cause data integrity errors – it may be impossible to write data, or reads may produce corrupt data. This affects the correctness of the processing.
- There are numerous ways and places that communication between the client component and the main processor may fail. If these fail at the wrong time during the completion of a transaction, the transaction may be incorrect; e.g., the main server completes the transaction, but the user client does not receive confirmation. The user thinks the transaction was not completed and tries again, ending up purchasing two tickets.

(Again, this is a sample only). In response to these modes of failure, we design measures to deal with them. These include measures to detect and report faults, recover from them, or prevent the faults from disrupting the system. Some of these measures are:

- Ping-Echo: In order to detect failed or unresponsive components so they can be restarted, a component sends out a periodic ping request to other components which must be answered within a certain timeframe.
- Raising Exceptions: Each component must detect certain types of error conditions and raise exceptions. (Handling the exceptions is of course also necessary, and other tactics are used to handle exceptions.)
- Active or passive redundancy: In order to minimize single points of failure, components are duplicated, with various different methods to ensure

synchronization between a failing component and its duplicate coming online to replace it.

- Transactions and checkpoint/rollback: Create atomic transactions that are recorded atomically, and ways to undo them if necessary.

Software patterns offer solutions to recurring problems in software design and implementation. A pattern describes a problem and the context of the problem, and an associated generic solution to the problem. Patterns have been used to document solutions to software problems. The best known software patterns describe solutions to object-oriented (OO) design problems [11], but patterns have been used in many aspects of software design, coding, and development [12].

Architecture patterns are patterns that describe proven architectural solutions to common system designs. They lay out the high-level structure and behavior of the system. They seek to satisfy multiple functional and non-functional requirements of the system. Several common architecture patterns have been developed, and are documented so that they can be widely used [5, 13, 14]. In this paper, we concern ourselves with architecture patterns and their relationship to reliability; for the remainder of this paper, all references to "patterns" refer to software architecture patterns.

One of the most common architecture patterns in the Layers pattern. The layered architecture consists of multiple hierarchical layers of functionality, each layer providing services to the layer above it, and serving as a client to the layer below [6]. In many systems, the lower layers are hidden from all except the adjacent higher layer.

The airline booking system's architecture is shown in the following diagram.

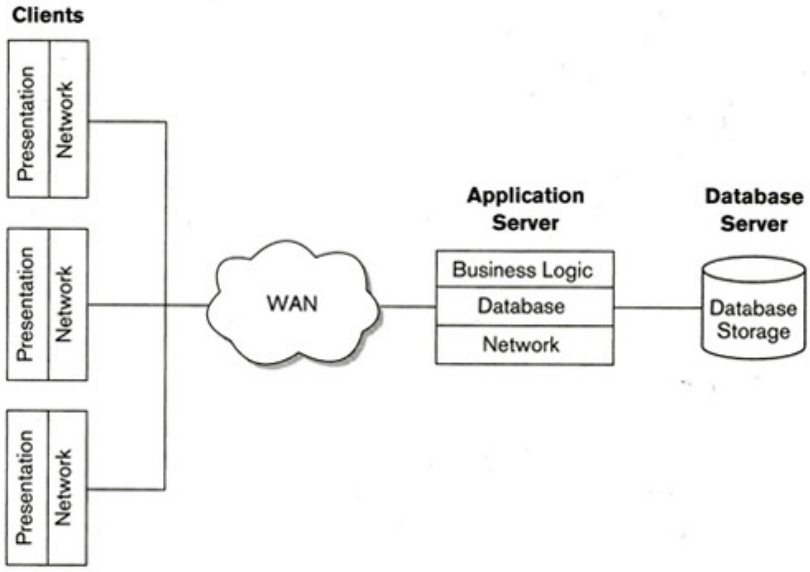


Figure 1: Airline Booking System Architecture

We can see the following patterns in the architecture:

1. Client-Server: The application server and database server are the server side, with multiple clients.
2. Layers: The application server exhibits a layered architecture.
3. Presentation Abstraction Control (PAC): The clients use the PAC pattern. Each client has a presentation component, which interacts with an abstraction of the system, and controlled by the business logic in the server.
4. Shared Repository: This pattern is quite speculative, as it is not clear whether accesses to the database are shared or not. For the purposes of this study, we err on the side of more complexity, and consider that the pattern is present.

3 Factors that Influence Tactic Implementation

There are three factors involved with the impact of tactics on a multi-pattern architecture. We summarize them as follows:

1. Tactics have a natural tendency to fall into one of three categories of impact on the components of the system. These categories are based on how broadly the tactic impacts the system, i.e. whether the tactics impact all or some of the system components. Most of the tactics affect only some of the components of the system. In this case, the key question becomes, "Which components are affected?" This is important, because implementation may be easy in some components (a good fit), and hard in others (high impact; a bad fit). Naturally, we want to implement a tactic in the easiest (low impact) way possible, but are there no guidelines for where it is easy or hard. Even if you know which components easily accommodate a tactic, that doesn't mean that you can automatically pick the easy spot. As noted above, this is influenced chiefly by the reliability requirements.
2. Previous decisions about the system become constraints to which the system must conform. Although all previous decisions may affect the selection and implementation of tactics, we are chiefly concerned in this work with decisions about software architecture patterns – mainly which architecture patterns to use. The documentation of many architecture patterns note whether there are particular problems with implementing certain reliability tactics. This helps answer the question about guidelines for where a tactic's implementation is easy or hard. In architectures with multiple patterns, you can then see where tactics fit well in the architecture.
3. Reliability requirements: There are two aspects of reliability requirements that are important. First, a requirement specifies a certain property of reliability to be achieved, such as high availability. This helps direct the selection of particular tactics to be used. The second aspect is that a reliability requirement must indicate what part(s) of the application it applies to. For example, a requirement of high availability specifies that it concerns call processing (not system administration). This directs where a component is to be implemented, namely in the part of the architecture that does call processing.

The next three subsections describe the factors in detail.

3.1 The Nature of Tactics

Section 2 has described how given tactic impacts a given pattern; additional detail is found in [8]. If a pattern is part of a multi-pattern architecture, the magnitude of the impact may change, but the nature of the impact of the tactic on the pattern does not change. For example, a common tactic for assessing the health of processes is "Ping-Echo." The nature of the impact is that processes must receive ping messages and respond to them. A multi-pattern architecture generally has more processes than a single-pattern architecture; thus more processes are impacted by the Ping-Echo tactic (all processes must receive and respond). Therefore, the magnitude of the impact of the Ping-Echo tactic is larger (than in a single pattern architecture), but the nature of the impact – the way that processes are modified – does not change.

The impact of a tactic on the individual patterns sets up the possible range of impact of the tactic on the entire system. In other words, we look at the impact of the tactic on the individual patterns in the system; the aggregate impact of the tactic on the system is generally no worse (greater) than the greatest impact on an individual pattern, and the impact is generally no better (less) than the smallest impact on any individual pattern.

Let us consider why. First, let us consider the best case: the tactic must be implemented somewhere in the system; the place with the least impact would be the pattern with the least impact; it can't be less than that. Even if one were to implement part of the tactic in one pattern and part in a different pattern, the amount of implementation can't go down; it's essential complexity (see [16]).

The worst case would be that the tactic must be implemented in all the patterns in the system. If the tactic is implemented in all patterns, it impacts each one, and the overall impact would approach the impact of the greatest impact on an individual pattern. This gives us a typical upper bound. For example, raising exceptions is typically required of all components of a system; therefore, it affects all the components of every pattern in the system. Therefore, each pattern feels the full impact of this tactic.

Let us now take a higher level view than detailed impact on individual patterns. We examine the impact of a tactic on the components of an architecture as a whole. In particular, we are interested in whether or not a tactic impacts a component. We find that there are three general categories of interaction of a tactic with the architectural components of a system. With one exception (the second category), we do not consider the details of that interaction, or how that interaction is accomplished. The categories are as follows:

1. **A tactic impacts all of the components** in the architecture. For example, a tactic for fault detection is Exception Raising. All components must implement Exception Raising.
2. **A tactic impacts all of the components** in the architecture, and one of the functions required by the tactic is that there is a **central coordinating component** that controls the other components in some way. (This is the exception about details of interaction that is mentioned above). For example, in the Ping-Echo tactic, a central process periodically sends requests to processes requesting that they verify their health.

3. **A tactic is implemented using just some of the components** in the architecture, leaving the remainder of the components unaffected. Which specific components are used depends on where the tactic is to be implemented, which is determined by the specific reliability requirements. For example, systems where the correctness of certain calculations are critical may employ the Voting tactic. In voting, the calculation is performed by three or more different components, each developed independently of the others. In this case, only the component that performs the calculation and its calling component are affected; all other components have no other changes.

We analyzed all the reliability tactics given in [2] and found that each tactic can be classified in one of the three above categories. We also analyzed several tactics from [3] and [4], and also found this to be true. Our evidence suggests that these categories are sufficient to classify all reliability tactics. Due to space limitations, we focus only on the tactics from [2].

A few words of explanation are in order. First, architectures are composed of connectors and components, as well as behavior. However we consider only components for simplicity. Also, based on previous experience [8], we expect that connectors work the same way as components. A thorough exploration of connectors and behavior is a subject of future work.

Second, one might wonder why the second category (impact all components, with a central controller) is a separate category from the first, after all, it is a special case of the first. The controlling component has strong architectural implications – a controller must communicate with all other components. If an architecture already has such a controller, it is often easy to incorporate the tactic, and may even be trivial. For example, the Broker pattern has such a component. On the other hand, if the architecture has no such component, adding it is usually very disruptive to the architecture. Therefore, it is useful to consider this category separately from the first.

In the airline reservation system, let us consider the tactic we identified. The tactics we identified fall into the following categories:

1. Ping-Echo: This requires that each component must respond to the ping messages. In addition, one component must initiate the ping messages, and manage the responses. (Category 2: impacts all components, plus a central component). (Another possible option is to dispense with a central controller, and have a scheme where components are responsible for monitoring each other's health.) It appears that the most natural central component is the application server, (see figure 2). Or because the application server may itself be composed of multiple software components (not shown), it may a component within the application server.
2. Exception Raising: Generally, Exception Raising should be done consistently across the application, which means that each component must raise exceptions as well as respond to other exceptions. (Category 1: There is no explicit central component, but all exceptions must be appropriately handled. This may hint at a central "handler of last resort", but it really depends on the tactics chosen to handle the exceptions). There are two special considerations: first, who should the database component report exceptions to? Clearly, exceptions should be reported to the component that can correctly handle the fault; in this case, it is likely the application server component. Second, should the client presentation components

also report exceptions to the application server? It is more likely that the exceptions be raised and handled locally.

3. Active Redundancy: A single component can manage the redundant components. Some systems are entirely redundant, while others may have a few critical components that are redundant; perhaps the data store or communication infrastructure components. Which components must be duplicated depends on the system requirements (Category 3). The obvious candidates for redundancy are the application server and the database. In order to maintain availability, the application server should be replicated, and active redundancy appears to be a viable choice. Because information from the database is necessary to make a reservation, the database should also be available at all times, and should be replicated.
4. Passive Redundancy: This impacts more than one component, because the passive component must receive state updates from the active. It is likely that the modifications can be confined to a few components though. (Category 3, impacts some components; who is duplicated depends on the requirements; same rationale as for Active Redundancy). Passive redundancy may be an alternative to active redundancy for the application server. Due to the fact that database actions are transactional in nature, passive redundancy may be a more natural choice for replicating the database than active redundancy. Note that regardless of the choice of redundancy type, one should try to design it so that it is invisible to other components; i.e., it should be entirely transparent to the clients.
5. Transactions: Processing is bundled into transactions which can be undone all at once, if necessary. Transactions make checkpointing and rollback easier. It affects those components that deal with the details of the transactions. (Category 3: impacts some components. The requirements help shape which components deal with transactions). Any database actions are naturally transaction-oriented; this is a good fit, and can be done entirely within the database component. However, a purchase is also a natural transaction, and the notion of transactions therefore must permeate the design of the application server, as well as the clients themselves. Here we see how the notion of a transaction is driven by the requirements, which in turn affects how and which components are affected.
6. Checkpoint/Rollback: Likely to affect all components that deal with the data/transactions/state that must be checkpointed and rolled back. This impacts some or all of the components that are directly involved with the state of the data being processed. (Category 3, because not all components are directly involved in the state of the system). This is a natural fit with transactions, and the impact follows the same pattern: database transactions can be checkpointed and rolled back entirely within the database components; rolling back of user purchases affects all components.

For each tactic, this information helps us understand the components needed for implementing it. Of course, this must be placed in the context of the architecture of the system, including the patterns used.

3.2 The Impact on Multiple Patterns

Because tactics are realized within the architecture, tactics have some effect on the architecture and the architecture patterns used. While the purpose of a tactic is to focus on a single design concern of reliability, the impact may be broad, affecting many or even all of the components of the architecture.

We have studied the impact of implementing tactics on patterns and in the case where the pattern provides the structures needed by the tactics , we found that the impact can be minimal. On the other hand, the impact can be great if the pattern's structures must be substantially modified, or if many different structures must be added. We described five levels of impact as follows:

1. Good Fit (+ +): The structure of the pattern is highly compatible with the structural needs of the tactic.
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.
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.
4. Significant Changes (-): The changes needed are more significant. They generally involve adding a few components that are not similar to the pattern, and/or moderately modifying the existing components.
5. Poor Fit (- -): Significant changes are required to the pattern in order to implement the tactic. They consist of the addition of several components, major changes to existing structure and behavior, and/or more minor changes, but to many components.

These levels describe the amount of impact on the architecture; full descriptions are given in [8]. One may also consider this to be a rough indicator of the difficulty of adding a given tactic, although we do not make any specifications of difficulty or expected effort.

In [8] we analyzed how a given reliability tactic is implemented in a given architecture pattern. We see that the tactic can require the architect to modify components and connectors, and possibly even create additional components and connectors. However, industrial systems are quite complex, involving multiple architecture patterns. Therefore, we must consider implementing tactics in this larger context.

In a study of the architectures of 47 industrial software projects, we found that 37 used more than one architecture pattern [15]. Most had two, three, or four patterns. The most we saw in a single architecture was eight. Therefore, it is not sufficient to consider the impact of a tactic on a single pattern, but we must consider the potential impact on all the patterns in the architecture.

There are several possibilities of how a tactic might interact with an architecture that contains multiple architecture patterns. A possibility on the one side is that the tactic interacts with all the patterns in the architecture. On the other extreme, the tactic might need to interact with only one of the architecture patterns. Clearly, the second possibility has a smaller impact on the architecture than the first. Therefore, for a

given tactic and a given system, the challenge is to determine how many of the patterns are impacted, and in what way the tactic implementation affects them.

The impact of tactics on multiple patterns is shaped by the category of tactic, as described previously. As the pattern category differentiates the tactics on their impact on components, and patterns embody components, we see how tactic categories relate to multiple pattern impacts. They are as follows:

1. If the tactic impacts all components, then it must be implemented in all the patterns. The magnitude of impact will tend to be at or near the magnitude of the "worst" pattern in the system. For example, the tactic "raising exceptions" requires that every component either raise exceptions or have a good reason not to. Thus this tactic affects every component of every architecture pattern.
2. If the tactic impacts all components and requires a central controller, then the impact on the system will often be better than the impact of the "worst" pattern. The reason is that high impact on the "worst" pattern may well be because one needs to add a central component (and all the associated connectors). However, if there is another pattern in the architecture that has a central component, then the tactic will probably be able to take advantage of it and can be implemented in that pattern. In fact, the necessary connectors will also probably be in place. Thus the impact on the system may be near the impact on the pattern with the central component. This impact can be quite low. So, in this case it depends on which patterns are present in the system. For example, during an architecture review of a distributed time-tracking system, we found that the designers had neglected to sufficiently handle cases where a client loses connectivity with the server. A heartbeat was added which (in this case) required that each component periodically generate a heartbeat message, as well as a central component to handle the heartbeat messages and detect unresponsive components. The system included the Broker and Layers patterns, and the central component was a natural fit with the broker component of the Broker pattern.
3. If the tactic impacts some of the components, one would certainly want to implement the tactic in the pattern with the smallest impact. However, depending on the requirements of the application, this may not always be possible. We discuss this in the next section. However, this gives a starting point for considering impact; a best case scenario. For example, in a space exploration simulation game, we explored the need to recover from erroneous input by using transactions and rollback. The system included both the Layers and Active Repository patterns. The designers could consider which of the patterns would be a best fit for these tactics, but the key consideration was the application itself – what exactly needed to be rolled back in the event of an error. In this application, the fact that games were dynamic indicated that transactions and rolling back were more appropriately implemented in the Layers pattern.

The following table summarizes the nature of the impacts:

Tactic Impact Category	**Impact on Patterns' Components**	**Impact Tendency**
All components	All components in all the patterns are affected	Impact is that of the pattern with the greatest ("worst")

		impact
All components, plus central controller	All components in all the patterns affected; placement of central controller is significant	If a pattern supports a central controller, impact is less than the "worst" pattern
Some components	May be possible to implement in a single pattern	Ideally, impact is that of the "best" pattern. But requirements play a major role here.

Table 1. Impact Categories' Impact on Multiple Patterns

This gives us a basic understanding how a given tactic will be implemented in the patterns of an architecture. It also gives us a basic idea of the magnitude of impact on the architecture caused by the tactic. However, this information is as yet insufficient to fully understand how a tactic will be implemented in the architecture. We need more information, specifically about the purpose of a given tactic in the system. We need to consult the requirements of the system; the other major factor.

In the airline reservation system, let us consider the impact of the tactics we identified on the patterns in the system. Remember the four patterns identified were Client-Server (CS), Layers (L), Presentation Abstraction Control (PAC), and Shared Repository (SR).

1. Ping-Echo: Impact: CS: +, L: +, PAC: ~, SR: ~. Since this tactic impacts all components, plus a central component, the overall impact should be neutral or better. Analysis: All components must be notified and respond. CS components have the necessary communication paths built in, and the Server component is a natural central component. Therefore this tactic is compatible with the patterns, but is not an ideal match (overall impact: +).
2. Exceptions: Impact: CS: ~, L: ++, PAC: ~, SR: ++. Since this tactic impacts all components, the overall impact tends to be the most severe of the patterns; in this case neutral.
3. Active Redundancy: Impact: CS: +, L: +, PAC: +, SR: ++. We see that this tactic is compatible with all the patterns. Since it impacts some components, the overall compatibility is good, and if the tactic can be confined to the SR pattern, the overall compatibility could be very good.
4. Passive Redundancy: Impact: CS: +, L: +, PAC: ~, SR: ++. The impact is similar to Active Redundancy, except for PAC, which is not as compatible. Does the fact that this tactic is not as good a fit with PAC as Active Redundancy push us toward Active Redundancy? Not necessarily. If the redundancy is implemented in components that do not interface with the PAC components, it doesn't matter.
5. Transactions: Impact: CS: ++, L: ++, PAC: +, SR: ++. This tactic is a good fit with all the patterns except PAC, where there are issues of keeping multiple presentations in synch. Since this tactic impacts some components depending on the requirements of the system, we may be able to implement it away from the PAC components.

6. Checkpoint/Rollback: CS: ++, L: ++, PAC: -, SR: ++. This is very similar to Transactions except that keeping multiple transactions in synch is likely more involved because of rollbacks. The analysis is the same, though.

3.3 The Role of System Reliability Requirements

The above general descriptions of impact are a starting point for understanding how a tactic impacts a system architecture. In addition, the system's reliability requirements that trigger the selection of the tactics play a major role in the impact of the tactics. Certain system requirements may cause a tactic to be implemented in certain components; that is, in certain architecture patterns. This can override any attempt to implement the tactic where it would be easiest to implement. Therefore, decisions about how (and where in the architecture) to implement tactics are driven not just by the architectural structure, but also by the system reliability requirements. This then demands that the detailed requirements be analyzed as part of the reliability architecting process.

System reliability requirements shape the implementation and impact of reliability tactics in two important ways. The first way is that reliability requirements influence which design concerns are to be satisfied by tactics. In particular, different ways in which faults affect the system are identified and the actions taken in response are decided in order to meet the requirements. For example, consider a telecommunications system that must be highly available to process calls – 99.999% of the time. In order to meet this requirement, architects identify that components may fail due to hardware or software failures. In order to meet the availability requirement, all components must run nonstop; therefore, failed components must be detected and restarted quickly. The design concern is timely detection of failed components. A tactic that implements this design concern might be Heartbeat.

The second way that requirements affect the impact of reliability tactics is that they often specify which part of the system a tactic applies to. For instance, in the example above, high availability applies to call processing only. Therefore, any components that are involved in call processing must implement their portion of the Heartbeat tactic. However, other components, such as those dealing with provisioning, system administration, or billing, will likely not be subject to the Heartbeat tactic.

The process of architecting is highly iterative and quite intuitive. Therefore, one doesn't necessarily determine all the reliability requirements first, etc. In fact, the requirements, architecture decisions, and tactical decisions are done in different orders, piecemeal, and basically together. So a decision to use a tactic, combined with a reliability requirement, might dictate the selection of a particular pattern that fits the tactic well. Or a tactic might be selected over a different alternative because it fits with a pattern that has already been selected. So the process is very fluid; we have found that consideration of architecture and the selection and design of reliability tactics often happen simultaneously [17].

Consideration of each of these factors helps complete the picture of how and where a tactic will be implemented in the architecture, as well as which tactics to use (if not already determined). The result is that the architecture is more complete: the

architecture patterns are now modified to accommodate the implementation of the tactics.

Let us consider the tactics in the airline reservation system that are category 3 – they impact some components based on the requirements of the system.

1. Availability: in considering Active or Passive Redundancy to help achieve high availability, one must decide which critical components must be replicated. The critical functionality to be replicated is the business logic, which is found in the Layers pattern, so the overall impact is that of Redundancy on Layers. Since the impact of both redundancy tactics on the Layers patterns is positive, it helps us understand that from an architecture viewpoint, it doesn't matter which redundancy tactic we select in this application.
2. In order to achieve data integrity, we consider using Transactions and Checkpoint/Rollback: Both tactics are a very good fit for all the patterns except PAC. So the question becomes whether transactions and rolling back can be defined below the level of the user interface. This is very likely – user interaction can be designed so that actions are encapsulated in transactions, and rolling back to previous states or data should be transparent to the user. This information guides us to design the user interaction along lines of transactions and gives us motivation for doing so.

4 Use in the Architecture Design Process

One of the major challenges in developing reliable systems is that decisions about implementing reliability must be made early; it is exceedingly difficult to retrofit reliability into an established architecture if it was not planned for. Yet the implications of architectural reliability decisions may not be understood, resulting in design and implementation difficulties later on. The information about how requirements, tactics and architecture patterns interact can help ameliorate these difficulties, or at least anticipate some of them. The information can be used during the architecture design process to consider tradeoffs in tactic selection, to refine and re-negotiate requirements, and to a lesser extent, in pattern selection or modification. The impact information is not intended to be used to generate specific effort estimates; it does not have sufficient detail or specificity.

In this chapter, we do not propose a specific architecture design process for incorporating reliability tactics into an architecture. Instead, we describe how the information about how requirements, tactics and architecture patterns interact can fit in a general model of architectural design [18]. The main activities in the model are architectural analysis, architectural synthesis, and architectural evaluation. The output of architectural analysis is architecturally significant requirements, the output of architectural synthesis is candidate architectural solutions, while architectural evaluation assess the candidate solutions with respect to the requirements.

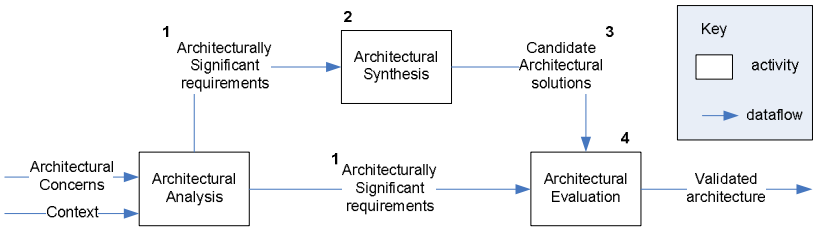
The first activity, architectural analysis, is focused on understanding the architecturally significant requirements. This includes refining reliability requirements and identifying the associated design concerns. For example, if high availability of the system is a requirement, an associated design concern may be

replication. A key question is which part of the system must run nonstop, as that will determine where the candidate tactics must be implemented. We might also ask whether the system may have momentary interruptions, which would allow a passively redundant solution rather than an active redundancy. The answers to these questions help clarify the requirements, and will be used (later) as tactics are selected and decisions made about where the tactics should be implemented.

The architectural synthesis activity is typically where tactics and patterns are considered. The design concerns point us to certain tactics; for example the replication design concern leads us to consider Active Redundancy, Passive Redundancy, and Spare. The candidate tactics and requirements are major drivers for pattern selection. This implies a certain amount of iteration among architectural analysis and synthesis, as the architecturally significant requirements and candidate patterns are iteratively refined. The information about the impact of tactics on multiple patterns can be used here to optimize the required effort from the combination of patterns and tactics. Candidate patterns and tactics are major pieces of candidate architecture solutions.

The architectural evaluation activity is to determine whether the proposed architectural solutions fulfill the architecturally significant requirements. The additional information about tactics' categories of interaction as well as the detailed reliability requirements can enhance the ability of architects to effectively evaluate candidate architectural solutions.

The following figure shows the architecture design activities as described by Hofmeister et al. It is annotated with the activities related to requirements, tactics and patterns, and their interactions. These are shown by the numerals attached to activities and data flows, and are described at the bottom of the figure.



1. Reliability requirements refined; design concerns identified.
2. Tactics and patterns identified, impacts identified, tradeoffs among tactics and patterns made. Requirements drive decisions of where to implement tactics.
3. Candidate solutions include tactic and pattern information, with impact information
4. Requirements and nature of tactics can help evaluate solutions

Figure 2: Architecture design activities from [18], with pattern and tactic activities

The following are two examples of how the tactic and pattern information is used in the architectural synthesis and evaluation activities to improve the quality of the architecture.

The activity was architectural synthesis, and the system performed automated sequential manipulation of paper. The key reliability requirements included that the papers had to be processed correctly, and that any machine malfunctions did not cause further problems (fault tolerance). Key tactics used included transactions, ping-echo, and exception raising. The main pattern was Pipes and Filters (P&F), which supported the sequential (and configurable) nature of the paper processing. Another prominent pattern was Model View Controller (MVC), which provided the user interaction with the system. The main challenge was that the reliability tactics all were poor fits for the P&F pattern. However, they all were able to take advantage of the controller component of the MVC pattern, which communicated with each of the components of the P&F pattern.

Discussion: reliability tactics and architectural patterns are generally considered simultaneously, so one might ask which came first. Did the architects tentatively select the patterns first, and then used their components in designing the reliability tactics, or did they see the need for a controlling component for the tactics, and then added the MVC pattern on top of that component? It is almost certainly some of both: the highly iterative nature of architectural design means that many ideas are considered and tried as the architects work to create an architecture. Regardless, the information about how tactics and patterns fit together help shape the architecture.

In the second example, the activity was an architectural evaluation, performed just as development was getting underway. The system was a distributed time tracking system. The key reliability requirements were data accuracy and availability, although availability was not critical – the consequences of downtime were not catastrophic. Key patterns were Client-Server and Layers (on the server); Broker was identified during the evaluation as a desirable extension to Client-Server. The review uncovered that the architects had not fully considered all the ramifications of faults, and as a result, the Heartbeat tactic was added. Impact on the three patterns showed that it was the best fit in the Broker pattern, which added to the motivation to adopt the Broker pattern. It should be noted that the reliability requirements indicated the need to know the health of the remote clients, which fit exactly with the Broker pattern.

Discussion: In this case, the designers were somewhat inexperienced in reliability; more experienced designers may well have designed the system more comprehensively for fault tolerance. In such cases, the reviews serve to highlight potential issues with design or maintenance of the software. For example, a review of the paper processing system in the first example revealed the incompatibility between the P&F pattern and the reliability tactics as a potential trouble spot during future maintenance and enhancement.

5 Validation and Case Study

Validation of this work has three parts that correspond to the three factors discussed in section 3: the nature of tactics, the impact on multiple patterns and the role of requirements. We began by analyzing the reliability tactics to determine how many components in a given architecture they affect (some or all). This confirmed the three categories of tactics interaction with components, described above. The second

part was to verify and refine the impact categories of tactics on multiple patterns, by considering the tactics applied to common pairs of architecture patterns. The third part was a case study that considered a real architecture where we identified its architecture patterns, and analyzed how the reliability requirements influence the selection and implementation of tactics.

5.1 Tactic Impact on Architectures

We analyzed all the reliability tactics from Bass [2]. For each tactic, we analyzed how it should be implemented: what functionality is needed, and what components and connectors are needed to implement that functionality. We determined whether the tactic's implementation must be in all components, or just some of the components.

We began by identifying how (in general terms) the tactic should work, and what components and connectors are needed. This was done by studying the tactic descriptions [mainly in 2, 3, and 4]. We then considered how the tactic would be implemented in a system. Would the tactic require that all major components of a system take part in implementing the tactic, or could the tactic be implemented in a subset of a system's components? In each case, we found that a tactic was clearly in one or the other category.

For the tactics that required implementation in all components, we also examined the tactic to see whether a central controlling component was a part of the tactic. We found that these tactics could be categorized into either needing a central component or not. Ping-Echo requires a central controlling component. Raising Exceptions does not (note that handling the exceptions is separate from raising them, and would be done using whatever tactics are most appropriate for the type of exception). The tactic "Heartbeat" requires implementation in all components, and may or may not employ a central controlling component. Aguilera et al describe the use of a heartbeat with no central controller in [19].

For the tactics that impact some components, we considered which components would be affected. The ideal model is that the tactic should be implemented in the pattern where it is the best fit – where the impact is the lowest. However, we found that the components where a tactic should be implemented depended on what part of the system needed the associated reliability. For example, there are several replication tactics (Active Redundancy, Passive Redundancy, and Spare). In order to decide in which pattern to implement the redundancy, one must decide which part of the system needs to be replicated. This would be dictated by the requirements of the system, namely what critical functions must run nonstop. We found that in every case where a tactic is implemented in some components, we could not say definitively where the tactic should be implemented, because it would provide that reliability feature to a particular part of the system. Instead, the answer was always, "It depends on the requirements to state which part of the system must have this reliability feature."

We found that each tactic fits into one of the three categories, as shown in the table below. (The categories, as described earlier are 1 – all components, 2 – all components with a central component, and 3 – some components).

Design Concern	Tactic	Impact Category
Tactics for Fault Detection		
	Ping-Echo	2
	Heartbeat	1 or 2
	Exceptions	1
Fault Recovery -- Preparation		
	Voting	3
	Active Redundancy	3
	Passive Redundancy	3
	Spare	3
Recovery -- Reintroduction		
	Shadow	3
	State Resynchronization	3
	Checkpoint/Rollback	3
Fault Prevention		
	Removal From Service	3
	Transactions	3
	Process Monitor	3

Table 3. Categories of Impact of Common Reliability Tactics

We see that most of these tactics impact some of the components. We also see that the tactics' categories appear to be generally consistent within design concerns. We have not studied other reliability tactics (from Utas [3], Hanmer [4], or other sources) enough to know whether these trends are consistent; this is noted as future work.

5.2 Impact of Tactics on Pairs of Patterns

To begin to validate the information about how tactics impact multiple-pattern architectures, we considered the impact of each tactic on common pairs of patterns. Future work is warranted to extend this to pattern triplets and beyond; however, in our analysis of the airline booking system (the running example), we found that the relationship among pairs applied sequentially appears to be the same as analyzing multiple patterns together. In our earlier work we showed that virtually all significant systems contain multiple architecture patterns [15], and that the most common pairs of architecture patterns identified were the following:

1. Broker – Layers
2. Layers – Shared Repository
3. Pipes and Filters – Blackboard
4. Client-Server – Presentation Abstraction Control
5. Layers – Presentation Abstraction Control

6. Layers – Model View Controller

We analyzed how each tactic would be implemented in a system consisting of each one of the aforementioned pairs of patterns. We determined whether the impact category (see Section 3.2) was valid and whether the nature and magnitude of the impact supported the descriptions given above. This analysis helped form and validate the categories and the nature of the impact of the tactics in each category. A summary of the impact is shown in table 3. In the following table, the type of impact of the tactic is given with the tactic name. In the boxes, the two impact ratings in parentheses are the ratings of the two patterns, respectively. The other rating is the composite rating. In many cases, the rating shows a range, or is given as "likely" or "close to." These are cases where the requirements play a major role in where the tactic should be implemented, and this affects the impact on the architecture.

Patterns / Tactics	Layers – Broker	Layers – Shared Rep	P&F – Blackboard	C-S – PAC	Layers – PAC	Layers – MVC
Ping-Echo (all, central)	(+, ++) ++	(+, ~) ~	(--, ~) Likely -	(+, ~) Close to +	(+, ~) ~ or better	(+, ~) ~
Heartbeat (all, central or not)	(+, ++) ++	(+, ~) ~	(--, ~) Likely -	(+, ~) Close to +	(+, ~) ~ or better	(+, ~) ~
Exceptions (all)	(++, +) +	(++, ++) ++	(--, --) --	(~, ~) ~	(++, ~) ~	(++, ~) ~
Voting (some)	(+, ++) + or better	(+, ~) Likely +	(++, +) Likely ++	(+, +) +	(+, +) +	(+, +) +
Act. Red. (some)	(+, ++) up to ++	(+, ++) likely ++	(++, +) + to ++	(+, +) +	(+, +) +	(+, +) +
Pass. Red. (some)	(+, ++) up to ++	(+, ++) likely ++	(-, ~) Likely ~	(+, -) Likely ~	(+, -) Close to +	(+, -) Close to +
Spare (some)	(~, ++) up to ++	(~, -) up to ~	(+, ~) Likely +	(+, ~) ~ to +	(~, ~) ~	(~. -) Likely ~
Shadow (some)	(+, ++) + or better	(+, ~) likely +	(+, -) - to +	(+, +) +	(+, +) +	(+, -) Close to +
State Resync (s)	(+, ++) +	(+, ++) + to ++	(--, +) Close to --	(+, -) Close to +	(+, -) Close to +	(+, ~) Close to +
Checkpoint Rollback	(++, ++) ++	(++, ++) ++	(--, --) --	(++, -) Close to	(++, -) Closer	(++, +) Close to

(some)				++	to ++	++
Rmve from Service (s)	(~, ~) ~	(~, ~) ~	(-, -) -	(~, ~) ~	(~, ~) ~	(~, ~) ~
Transactions (some)	(++, ++) ++	(~, ~) ~	(-, --) Close to -	(++, +) Close to +	(++, +) Close to ++	(++, ~) Close to ++
Process Monitor (s)	(++, ++) ++	(~, ~) ~	(-, -) -	(~, ~) ~	(~, ~) ~	(~, ~) ~

Table 4. Impact of Tactics on Pairs of Patterns

We note that tactics that are category 3 (some components) normally have impact that ranges between the impacts of the two patterns; the impact depends on the reliability requirements. However, in cases where the impact of the two patterns is the same, there would be no difference so we do not see a range of impact. We see this in several of the tactics in the table.

In the table we see that the Broker-Layers pattern pair is most compatible with the tactics. In fact, only one tactic has worse than a positive impact. The Broker-Layers pair is also the most common pair we found. This is more that good fortune: architecture patterns are usually at least partly selected based on the tactics that are selected (see [2]). We can surmise that one reason for selecting the Broker and Layers patterns is to accommodate one or more of these tactics.

5.3 Case Study: Review of an Architecture

We performed an architectural review of a system. As part of this review, we identified the patterns in the architecture, the tactics used to achieve high reliability, and how the tactics and patterns interacted. The data from this review should support or refute the following questions:

1. Do the reliability tactics used impact multiple patterns?
2. Do the tactics impact the patterns in the tree ways described?
3. How do the failure modes impact where tactics are implemented?

The system we reviewed is proprietary, so details that identify the company, exactly what the system processes, and the exact architecture cannot be given. A general description of the system is as follows: It provides customized sequential processing of certain types of physical materials. It is in effect, an automated assembly line, with sequential stages, performing actions on the materials. The system includes custom hardware modules, controlled by software within each module, as well as central control of the entire assembly line.

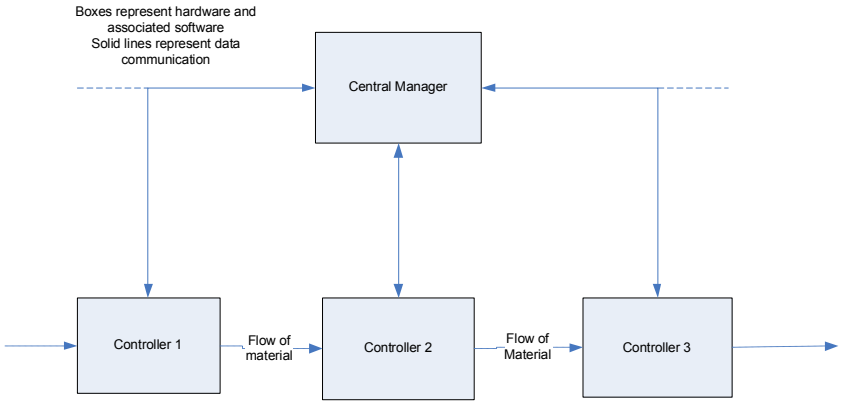


Figure 2. Generalized architecture of assembly processing system

The system has important reliability requirements. The most important are that the assembly must be done correctly – they must guarantee that no finished product has been assembled incorrectly. A closely related requirement is that no finished product may have any damaged parts. Another important requirement is that the system must have high throughput; however, this does not imply that high availability is required.

Important failure modes, as well as the measures (tactics) adopted by the system to deal with them included the following:

1. A hardware module ceases to function because of a hardware or software malfunction. In order to detect this, the designers used a Ping-Echo, with a central controller. Corrective action included notification of upstream modules to suspend work, but allowed downstream modules to complete processing. This is roughly analogous to the tactic, "Fail to a Stable State", described by Utas [3]. Repairing the unit was a manual operation, so an alert was issued to the user.
2. Materials may be damaged by processing, or may arrive already damaged. In any case, a module may receive damaged materials. The modules have no way of automatically discarding damaged materials, so the corrective actions are the same as number 1. The difference is in detection: a module can detect damaged materials and use the "Raise Exceptions" tactic to inform the central controller.
3. If the communication link between the central controller and a processing module fails, the module may not be able to respond to commands such as suspending processing, nor can it report faults such as damaged materials. In this case, it appears to the central control that the module is not responsive, so it treats it as number 1, above.
4. The result of suspending processing can result in materials being not completed, or perhaps being completed incorrectly. The system must be designed so this does not happen. In order to prevent this problem, the processing of materials was divided into discrete units that could be completed independently; these units can be considered to be transactions of work.

The architects used the following tactics to achieve their reliability goals: Ping-Echo, Raising Exceptions, Fail to a Stable State, and Transactions. The key feature of

the architecture was a set of independent hardware modules, arranged in sequence to process the materials. Their operation was coordinated by a central coordinator, which included a user interface. The architecture used numerous patterns, notably Model-View-Controller (the View was the user interface, the central controller was the Controller, and the processing modules together were the Model), Pipes and Filters (the Filters were the processing modules), Layers (within each processing module), and State-Driven (the system taken together).

Let us see how each tactic supports or refutes the earlier questions.

1. Ping-Echo must be implemented in the processing modules, but requires a coordinator. It does impact multiple patterns: each filter in Pipes and Filters must implement it; within each, at the highest layer in the Layers pattern; the State-Driven system must be aware of it, and the Controller in Model-View-Controller coordinates the pings.
2. Exceptions are raised by the Pipes and Filters, and the Layers within them. Any components involved with the system state would raise exceptions if there are any issues with state. Since the Filters are also the Model, they raise exceptions, but more to the point, the Controller must have some mechanism for catching the exceptions. So all patterns are affected.
3. Fail to a Stable State was not listed in the main analysis of the tactics, but it clearly impacts the Model-View-Controller, the Filters, and the State-Driven patterns.
4. Transactions impact the Filters and perhaps the Layers within them. Since the concern is the unfinished work within the Filters, it may be possible for the Filters to handle this tactic without involving other patterns – for example, the controller may simply have to issue a "resume" command, and the Filters complete the transactions in progress.

The impact of the tactics used on the architecture is shown in the following table. This table shows the impact of the tactics on the individual patterns, and the overall impact, along with an explanation.

	Pipes & Fiters	Layers	MVC	State-Driven	Overall
Ping-Echo	- - (each Filter must respond, needs central cntl)	+ (good fit)	~	~ (States and pings orthogonal	- - (each Filter must respond; MVC provides cntl)
Raise Exceptions	- (each Filter must raise exceptions	++ (also is natural fit for handling)	~	+ (also good fit for handling)	- (all components must implement, including Filters
Fail to Stable State	+ (Filters can simply stop processing)	++ (Layers can catch lower level errors)	+ (States mainly in Model)	++ (a natural fit)	+ (all components affected, including Model and Filters)

Transactions	~ (Can help to divide work)	++ (good fit)	~	++ (a natural fit)	++ (implemented in State: in Model and Filters with few changes to them)

Table 5. Impact of tactics on individual patterns and overall architecture

This shows us two characteristics of implementing tactics in the architecture. First, we see that some tactics might be implemented where there is a good fit with the patterns in the architecture. We see this with the following tactics: Transactions, and Fail to a Stable State. Of course, this depends on the types of failures and how they must be handled according to the requirements.

The second characteristic is that some tactics require that all the components of the software implement the behavior of the tactic; this is the case with these tactics: Exceptions and Ping-Echo. The impact of this characteristic is particularly striking in the case of Ping-Echo: in order for the filters to implement it, they had to establish direct communication with a central component, as well as implement mechanisms to respond to the ping messages in a timely manner. This caused a significant deviation from the standard Ping-Echo pattern.

This case study shows an example of each of the three types of impact of tactics described earlier. It shows how these tactics impact an architecture consisting of multiple tactics.

6. Related Work

The reliability tactics originally described by Bass et al [2] have been explored in more depth. Several of the tactics have been further specified, resulting in new sets of more specific tactics [22]. For example, the tactic called "Raising Exceptions" has been subdivided into tactics of "Error Codes" and "Exception Classes." While we have not examined these newer tactics in depth, we expect that these tactics have the same characteristics as their "parent" tactics, and have the same architectural impact. For example, the two exception tactics cited above are alternate ways of implementing raising exceptions below the level of the architecture; the architectural constructs for both are the same.

Tekinerdogan et al discuss using failure scenarios in software architecture reliability analysis as a way of identifying candidate tactics for improving the systems' reliability [23]. This identifies what must be implemented; this work adds information about how such tactics can be incorporated into the architecture, and the impact on the architecture of doing so.

Reliability is an important topic in software architecture evaluations; important issues identified during architecture reviews and evaluations are often associated with reliability (see 24, 25, 26, 27]). A part of assessing the risk of reliability issues, one

should consider the impact of impacting their fixes – the tactics. This work helps architects understand the impact, and can thus help architects make more informed decisions during reviews.

Significant work has been done to analyze and predict reliability of systems based on the software architecture [28]. Approaches include using reasoning frameworks to do so [29, 30]. On the other hand, this work focuses on the impact on the architecture of measures taken to improve reliability. These are compatible; both should be considered when analyzing an architecture for reliability. A general reasoning framework for designing architectures to achieve quality attribute requirements has been proposed by Bachmann et al [31]. In this model, the impact of tactics on the architecture can be one of the inputs to the reasoning framework.

7. Future Work

We have studied a few of the tactics found in [3] and [4]; initial analysis supports the tactic categories and impacts shown here. All these tactics, as well as others found, should be studied. Producing a catalog of known reliability tactics along with their impacts would be useful. Such a catalog will need widespread input, as well as continuous updating.

We have observed that the categories of impact for the tactics tend to be similar for patterns that address the same design concern. (The design concern of fault detection has two in category 1 and one in category 2, but all affect all the components). It may be that the design concern influences or even dictates the tactic's impact category. However, only the Bass tactics are classified by design concerns. In order to determine whether this is a general rule, one must first classify other reliability tactics by design concern. Nonetheless, this appears to be potentially interesting, and we intend to study it further.

Some patterns and reliability tactics fit particularly well together, and may indeed be commonly used. We would like to investigate architectures to see whether some combinations of patterns and reliability tactics are common. These may form a set of "reliable architecture patterns;" variants of architecture patterns especially for highly reliable systems.

One very interesting consequence of implementing the tactics is that since it involves changing the architecture, some changes may actually change the pattern composition of the architecture. An architecture pattern may be added. In certain cases, an existing pattern may even change to a different pattern. Obviously, the transformation of one pattern to another can happen only where patterns are similar. We have seen two examples of this type of transformation in architectures we have evaluated. We intend to study this further in order to understand its architectural implications.

Further study should be done to examine the impact of reliability tactics on each other, and on other quality attributes, such as performance. Work has been done on tradeoff analysis as part of architectural analysis [10, 31]. Studies of reliability tactic interaction can provide specific information as input to such tradeoff analyses.

8. Conclusions

Measures taken to improve reliability (reliability tactics) are implemented in the context of three factors that influence its impact on the architecture of the system:

- The reliability requirements, which strongly influence which tactics are to be used, and what part of the system they apply to.
- Characteristics of the tactics themselves, namely whether the tactic has a natural tendency to be applied to all components of the system, or just a selected part.
- Constraints from other requirements and from design decisions. In particular, the architecture patterns used are important factors, because architecture patterns are commonly used, and the tactics impact them in regular and known ways.

Taken together, these factors create a picture of the impact of tactics on non-trivial architectures; those that involve multiple architecture patterns. This is of practical application, as most industrial systems use multiple patterns in their architectures. Architects can leverage this information to understand the potential impact of tactics on an existing or proposed architecture. They can use this to help make tradeoffs concerning the architecture and reliability tactics being used.

We have examined how reliability tactics would affect a real architecture, and found that the factors described above affect the impact of the tactics on the architecture as expected. We have also proposed how the investigation of the impact of tactics can be incorporated into typical software architecting processes. We recommend that this information be used during architecture of highly reliable software systems.

References

1. International Standards Organization, Information Technology – Software Product Quality – Part 1: Quality Model, ISO/IEC FDIS 9126-1.
2. L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison-Wesley, 2003.
3. G. Utas, Robust Communications Software: Extreme Availability, Reliability and Scalability for Carrier-Grade Systems, Wiley, 2005.
4. R. Hanmer, Patterns for Fault Tolerant Software, Wiley Software Patterns Series, Wiley, 2007.
5. F. Buschmann, et al., Pattern-Oriented Software Architecture: A System of Patterns, Wiley, 1996.
6. M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Addison-Wesley, 1996.
7. M. Shaw, Toward Higher-Level Abstractions for Software Systems, Tercer Simposio Internacional del Conocimiento y su Ingerieria, (Oct 1988) 55-61. Reprinted in *Data and Knowledge Engineering*, 5 (1990), 19-28.
8. N. Harrison, P. Avgeriou, Incorporating Fault Tolerance Techniques in Software Architecture Patterns', International Workshop on Software Engineering for Resilient Systems (SERENE '08), Newcastle upon Tyne (UK), 17-19 November, 2008, ACM Press.
9. N. Harrison, P. Avgeriou, Leveraging Architecture Patterns to Satisfy Quality Attributes, First European Conference on Software Architecture, Madrid, Sept 24-26, 2007, Springer LNCS.

10. W. G. Wood, A Practical Example of Applying Attribute-Driven Design (ADD), Version 2.0, Technical Report CMU/SEI-2007-TR-005, Software Engineering Institute, 2007.
11. E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading MA, 1995.
12. N. Harrison, P. Avgeriou, U. Zdun, Architecture Patterns as Mechanisms for Capturing Architectural Decisions, IEEE Software, 24(4) 2007.
13. D. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Distributed Objects, Wiley, 2000.
14. P. Avgeriou, U. Zdun, Architectural Patterns Revisited – a Pattern Language, 10th European Conference on Pattern Languages of Programs (EuroPLoP), (2005).
15. N. Harrison, P. Avgeriou, Analysis of Architecture Pattern Usage in Legacy System Architecture Documentation, 7th Working IEEE/IFIP Conference on Software Architecture (WICSA), Vancouver, 18-22 February 2008, 147-156.
16. F. P. Brooks, No Silver Bullet—Essence and Accident in Software Engineering, IEEE Computer, 20(4), April 1987, pp 10-19.
17. N. Harrison, P. Avgeriou, U. Zdun, Focus Group Report: Capturing Architectural Knowledge with Architectural Patterns, 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006), Irsee, Germany.
18. C. Hofmeister, P. Kruchten, R. L. Nord, H. Obbink, A. Ran, P. America, Generalizing a Model of Software Architecture Design from Five Industrial Approaches, 5th Working IEEE/IFIP Conference on Software Architecture (WICSA), November 06 - 10, 2005, IEEE Computer Society, 77-88.
19. M. K. Aguilera, W. Chen and S.Toueg, "Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks" *Theoretical Computer Science*, special issue on distributed algorithms, 220:1, June 1999, 3-30.
20. N. Rozanski and E. Woods, Software Systems Architecture, Addison-Wesley, 2005.
21. G. Booch, Handbook of Software Architecture: Gallery, http://www.handbookofsoftwarearchitecture.com/index.jsp?page=Blog accessed 4 February 2010.
22. J. Scott and R. Kazman. Realizing and Refining Architectural Tactics: availability, Technical Report CMU/SEI-2009-TR-006, Software Engineering Institute, 2009.
23. Tekinerdogan, B., Sozer, H., and Aksit, M. 2008. Software architecture reliability analysis using failure scenarios. *J. Syst. Softw.* 81, 4 (Apr. 2008), 558-575. DOI= http://dx.doi.org/10.1016/j.jss.2007.10.029
24. L. Bass, et al, Risk Themes Discovered Through Architecture Evaluations, Technical Report CMU/SEI-2006-TR-012, 2006, Software Engineering Institute, 2006.
25. G. Abowd et al., Recommended Best industrial Practice for Software Architecture Evaluation, Technical Report CMU/SEI-96-TR-025, Software Engineering Institute, 1997.
26. J. Maranzano, et al., Architecture Reviews: Practice and Experience, IEEE Software, 22(2) 2005, 34-43.
27. P. Clements, R. Kazman, M. Klein, Evaluating Software Architectures: Methods and Case Studies, Addison-Wesley, 2002.
28. Gokhale, S. S. 2007. Architecture-Based Software Reliability Analysis: Overview and Limitations. *IEEE Trans. Dependable Secur. Comput.* 4, 1 (Jan. 2007), 32-40. DOI= http://dx.doi.org/10.1109/TDSC.2007.4
29. T. Im and J. D. McGregor. Toward a reasoning framework for dependability. *DSN 2008 Workshop on Architecting Dependable Systems*, 2008.
30. L. Bass et al. Reasoning Frameworks, Technical Report CMU/SEI-2005-TR-007, Software Engineering Institute, 2005.
31. Bachmann, F, et al., Designing software architectures to achieve quality attribute requirements. IEE Proceedings 152(4), (2005), 153-165.