On the Impact of Fault Tolerance Tactics on Architecture Patterns

Neil B. Harrison
University of Groningen
Utah Valley University
800 West University Parkway
Orem, Utah 84058 USA
+1 801 863-7312
neil.harrison@uvu.edu

Paris Avgeriou
University of Groningen
PO Box 407
9700 AK Groningen, the Netherlands
+31 50 3237057
paris@cs.rug.nl

Uwe Zdun
Vienna University of Technology
Information Systems Institute
Argentinierstrasse 8/184-1
A-1040 Wien, Austria
+43-1-58801-58406
zdun@acm.org

ABSTRACT
One important way that an architecture impacts fault tolerance is by making it easy or hard to implement tactics that improve fault tolerance. Information about how the implementation of fault tolerance tactics affects the architecture patterns of a system should be useful to architects during architectural design in selecting optimal fault tolerance tactics and architecture patterns. In order to understand more about how useful this information can be, we performed an informal study of teams designing fault tolerance tactics in an architecture. One group used information about the interaction of tactics and architecture patterns; the other did not. We observed that the group with the information produced better quality architectures, and were able to better estimate the difficulty of implementing the tactics. We recommend that information about the interaction of tactics and architecture patterns be made available to architects, particularly those with less familiarity about fault tolerance tactics.

Categories and Subject Descriptors
D.2 [**Software Engineering**]; D.2.11 [**Software Architectures**]: Patterns; D.4.5 [**Reliability**]: Fault-tolerance

General Terms
Reliability

Keywords
Patterns, Software Architectures, Fault-tolerance, Reliability tactics

1. INTRODUCTION
Fault tolerance is not an afterthought. The design of fault tolerance measures must be undertaken early, because fault tolerance it tightly linked to the architecture of the system. Indeed, the architectural components and connections among those components may be highly compatible with certain fault tolerance measures, or may be of no help whatsoever. Thus the architecture selected influences the ease with which one can implement certain fault tolerance measures.

Conversely, measures taken to improve fault tolerance can be implemented with little or no change to the architecture, or may require significant changes: addition of extra components and connectors, or major modifications to the ones that are there. Such changes, besides requiring significant effort to implement, can obscure the original intent of the architecture.

Thus the architect faces the challenge of designing a fault-tolerant system architecture. Ideally, the architecture supports the fault tolerance measures, and they are implemented within the architecture. However, reality always intervenes: other functional and non-functional requirements also shape the architecture, resulting in an architecture that does not perfectly support fault tolerance. This forces the architect to make tradeoffs; to select fault tolerance measures that are most compatible with the architecture, or modify the architecture to accommodate fault tolerance.

In order for architects to make these decisions correctly and efficiently, they need information about how fault tolerance and architecture interact. Some work has been done that identifies how common fault tolerance measures (called tactics) impacts common architectural structures (called architecture patterns). We found that implementing these tactics affects patterns by modifying their components, adding components and connectors, and/or replicating components and connectors [16]. We identified the interactions of several of the most common fault tolerance tactics with several of the most common architecture patterns. The question now is how useful this information can be to architects.

To this end we have begun to study how the information about fault tolerance tactics and software patterns can be used to help architects make proper decisions about system design. This paper describes a study we conducted to give us more insight into the topic. The research problem addressed in this study is how can the knowledge of the interaction of fault tolerance tactics and architecture patterns help architects to make choices to optimally incorporate fault tolerant measures in the system architecture? In order to answer these questions we performed an exploratory study that is similar to a case study [25], although less formal In this study we compared how architects design with tactic-pattern interaction information against design without that information.

SERENE 2010, April 13-16, 2010, London, UK.

The rest of this paper is organized as follows: Section 2 gives background information about fault tolerance tactics and software architecture patterns. Section 3 describes our research questions. Section 4 describes the case study design and execution. Sections 5 and 6 give the results and their interpretation. Section 7 describes conclusions and future work.

2. BACKGROUND

The key concepts in this study are fault tolerance tactics and software architecture patterns, and their interaction.

2.1 Fault Tolerance Tactics

Bass et al. [2] define measures to improve quality attributes as tactics. There are two different types of tactics, designated as design time and runtime tactics. Design time tactics are measures that are applied across all parts of the system at design and coding time. They often take the form of design or coding rules, such as "check all return codes," or "prevent buffer overruns." Each developer must apply these tactics when designing and writing code. In contrast, runtime tactics are specific actions the system will take to achieve the desired quality attribute while the system is running.

An important set of runtime tactics are those to improve the fault tolerance of the system. In particular, the system takes certain actions to detect faults and errors in the running system, prevent faults from impacting the integrity of the system, and recovering gracefully from faults if they do occur. Typical examples of fault tolerance runtime tactics include voting and rollback. Bass et al. describe four categories of well-known fault tolerance tactics. Within a category, the tactics are often alternatives to each other.

1. Fault Detection: Measures to detect faults, including incorrect actions and failure of components. The tactics are Ping/Echo, Heartbeat, and Exceptions.
2. Recovery – Preparation and Repair: preparing for recovery actions; in particular redundancy strategies so that processing can continue in the face of a component failure. The tactics are Voting, Active Redundancy, Passive Redundancy, and Spare.
3. Recovery – Reintroduction: Tactics for repair of a component that has failed and is to be reintroduced. The tactics are Shadow, State Resynchronization, and Rollback.
4. Prevention: Preventing faults from having wide impact by removing components from service or bundling actions together so they can be easily undone. The tactics are Removal from Service, Transactions, and Process Monitor.

Full descriptions of these tactics can be found in [2]. We worked with these tactics because they are well known, and are described and organized conveniently. There are other fault tolerance techniques similar to tactics, such as are found in [12] and [24]. Techniques for specific aspects of fault tolerance have been proposed, including exception handling [8][10][11][18]. A comprehensive list of works concerning architecting fault tolerant systems can be found in [22].

In this paper, we refer strictly to the runtime fault tolerance tactics in Bass et al. Throughout the rest of this paper, we will refer to them as "tactics."

Tactics are implemented much like features: each tactic has a design, and is generally decomposed into components, connectors between the components, and required behavior. Thus it follows that the structure and behavior of a tactic impacts the structure and the behavior of the system. This is an important point at which fault tolerance (implemented via tactics) and the architecture meet.

2.2 Software Architecture Patterns

Software patterns are proven solutions to software problems, in a given context [9]. Architecture patterns are common architectural structures, which are well understood and documented [4][23]. These patterns describe the high level structure and behavior of systems. Architecture patterns describe the major partitions of a system in terms of components and the connectors between them.

Many common architecture patterns are described in [1][4][5][23]. Common architecture patterns include Shared Repository, Layers, Pipes and Filters, Presentation Abstraction Control, Model View Controller, Broker, Client-Server, and State Transition.

In the study described in this paper, participants worked with a system that employed the Broker and Pipes and Filters patterns; short descriptions follow:

- The Broker pattern structures distributed software systems with decoupled components that interact by remote service invocations. A broker component coordinates communication such as forwarding requests [4].
- The Pipes and Filters pattern structures systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters [4]. Filters do not know the identity of their upstream or downstream filters [23].

During architectural design, an architect may select one or more architecture patterns to follow to produce a system structure. The architect selects patterns based on their ability to support the requirements of the system, including fault tolerance requirements. Nearly all non-trivial systems employ more than one architecture pattern in their architecture [13].

Patterns embody the high level structure and behavior of the system. The structure and behavior of tactics is more local and low level, and therefore must fit into the larger structure and behavior of patterns applied to the same system.

2.3 Patterns and Tactics

The implementation of tactics in a system must, of course, be done within the context of the architecture patterns used in the system. In other words, architecture patterns describe the major components of the system and their relationships with each other (which involves the connections between the components and their behavior with respect to each other). But the implementation of tactics also involves components, connections between the components, as well as behavior. These may be to a greater or lesser degree compatible with those of the architecture patterns of the system [14].

For example, the Ping-Echo tactic is commonly used for detecting whether processes are operating sanely. Bass et al.

describe Ping-Echo as follows: "One component issues a ping and expects to receive back an echo, within a predefined time, from the component under scrutiny" [2] It is common that a single component monitors the sanity of several components; i.e., a central component is responsible for the health of much of the rest of the system. (This allows a central component to kill and restart any non-responsive processes.) This requires a connector between the central component and each component under scrutiny.

The Broker pattern has a central component which has connectors to each service component. This component can easily assume the role of the central component for Ping-Echo. In addition, the necessary connectors to components under scrutiny are already in place. Thus we find that the Broker pattern is a very good match for implementing the Ping-Echo tactic.

On the other hand, let us consider implementing Ping-Echo in the Pipes and Filters architecture pattern. Pipes and Filters consists of components that are connected and executed sequentially, with one-way communication from one component to another. The components operate autonomously; there is no central control. In order to implement Ping-Echo, a new component must be introduced. In addition, new connections must be provided to each of the Filter components. This also causes a change in the Filters' behavior: they must now respond to asynchronous messages from the control component in a timely manner. These changes mean that significant work is required to implement Ping-Echo in this pattern; Ping-Echo is a particularly poor fit for the Pipes and Filters pattern.

Because systems typically include more than one architecture pattern, one must consider all the architecture patterns when implementing a tactic: the tactic may impact some or all of the patterns. Architects must determine how and where in the architecture a tactic should be implemented in order to achieve the desired fault tolerance capability.

Tactic implementation can impact architectural patterns in a number of ways. Table 1 from [16] shows the types of impact on a pattern's components, sorted from low to high impact.

Table 1. Types of changes to pattern components

Type of Change	Description	Impact
Implemented in	Part of the tactic is implemented within a component, with no external change to the component. (A special case of Modify)	Only the behavior of the component changes. Generally the easiest to implement.
Replicates	A component is duplicated, with little or no change to its behavior. Usually done for redundancy. (Specialization of Add.)	Usually easy to implement.
Add, in the Pattern	A new component is added within the structure of the pattern (e.g., a layer is added in the Layers pattern.)	Generally easy or moderately easy to implement.
Add, out of the Pattern	A new component is added that is not part of the pattern structure, causing the system to deviate from the original pattern (e.g., adding a monitor to Pipes and Filters.)	Usually difficult to implement. Makes the pattern difficult to find, making maintenance more difficult.
Modify	The behavior and the structure of the component changes.	Impact varies: some changes are trivial, but others are very difficult.

The impact of a tactic on the connectors is similar: it can use the connector with no modifications, replicate the connector, modify the connector or require completely new connectors to be added.

Other examples of architecture patterns modified by the implementation of fault tolerance tactics include the Layers pattern [21], and the C2 architecture pattern [7].

The challenge to architects is to incorporate, in the system design, the tactics needed to achieve the desired levels of fault tolerance within the constraints of the architecture patterns. This involves making tradeoffs by selecting one or more of the following options:

1. Alternate tactics to achieve the same fault tolerance goal may be selected to better fit in the architecture.
2. In some cases, mainly early in architectural design, a different architecture pattern may be selected that better accommodates the desired tactics.
3. Within an architecture that has multiple patterns, one will naturally attempt to implement the tactic where it best fits – where the components and connectors of the pattern(s) can support the tactic with minimal changes. This, however, is constrained by how the tactic will be used. For example, a redundancy tactic may replicate a component of a particular pattern; in this case the architect cannot choose which pattern to use to implement the tactic.
4. Where no tradeoffs of patterns or tactics can be made, one must understand how the tactic will be implemented, even if it is not a good match. This helps one understand the implementation of the tactic. It can also help future developers understand why the architecture patterns will have been changed – to accommodate the tactics.

In order for an architect to make these tradeoffs, the architect uses information about how the desired tactics are implemented in the architecture patterns – the interaction of the tactics with the patterns. We designate this as "tactic-pattern interaction information." For a given tactic and pattern, it gives an estimated degree of implementation difficulty (on a five-point scale as described in [16],) as well as a description of how the components and connectors of the pattern must change to implement the tactic.

The tactic-pattern interaction information for the Broker and Pipes and Filters patterns, two patterns that figure prominently in this paper, are attached in an appendix.

3. RESEARCH QUESTIONS

The research question is whether the knowledge of the interaction of fault tolerance tactics and architecture patterns help architects to make choices to optimally incorporate fault tolerant measures in the system architecture? More specifically, we questioned whether the tactic-pattern interaction information supports the architects in:

1. choosing tactics that satisfy the fault tolerance requirements
2. designing the tactics correctly within the patterns in the architecture? In other words are the necessary components and connectors added or modified, and are there any missing
3. choosing tactic-pattern combinations that have the least impact on the architecture
4. understanding the effort required to implement the tactics

4. STUDY DESIGN

We wished to gain insight about these questions from practitioners in order to begin to move beyond abstract speculation and to set the stage for future empirical research on the topic. Therefore, we performed exploratory research as described by Kitchenham et al [20]. Our study venue limited the size and composition of the participants, so we performed an informal study that is similar to a case study as described by Wohlin et al. [25]. We compared the effectiveness of architectural design using tactic-pattern interaction information against design without that information.

The study compared two groups of participants. The participants were a group of attendees at the European Conference on Pattern Languages of Programming (EuroPLoP) who agreed to participate in the study. The ten participants were randomly divided into two teams, a control team and a study team. The participants did not know the purpose of the study or whether they were placed in the control or the study group. A check of the participants' experience revealed that the experience with software development, architecture, patterns, and fault tolerance were roughly equal in each group. Participants had considerable development experience, and were highly familiar with patterns. They had limited experience with architecture patterns, and had low experience with fault tolerant software. Table 2 summarizes the experience of the participants.

Table 2. Participant Demographics

	Experience: Industry/ Academia (years)	Architecture Experience	FT Experience
Study Team	11 / 0	Strong	Little
	10 / 0	Moderate	None
	0 / 2	Some	Little
	5 / 8	Moderate	None
	1 / 4	Little	Little
Average	5.4 / 2.8	Moderate	Little
Control Team	4 / 6	Some	Little
	3 / 2	Some	Little
	3 / 3	Little	Little
	9 / 0	Strong	None
	4 / 14	Strong	Little
Average	4.6 / 5.0	Moderate	Little

Each team was given a specification of a hypothetical system along with an initial architecture. The system represented control of an automated manufacturing assembly line, and involved managing numerous distributed robotic devices. Control moved sequentially through the robots (Pipes and Filters architecture pattern), while supplies were managed through an application of the Broker pattern.

The teams were given four tasks to add specific fault tolerance measures to the system. They were asked to design the fault tolerance, and show the resulting architecture. The tasks were to be performed in order. The fault tolerance tasks involved the tactics from the following categories:

1. Detection of a failed component (Tactic: Ping-Echo)
2. Replacement of a failed component (Tactic: Spare)
3. Recovery from erroneous actions (Tactic: Transaction-Rollback)
4. Detection and replacement of a failing component (Tactics: Process Monitor and Passive Redundancy)

Other tactics might be used, such as Heartbeat instead of Ping-Echo.

The teams were not told which tactics to use, or even which category of tactics to use. However, each team was given descriptions of all the fault tolerance tactics from [2]. They had to choose which tactic best satisfied the task, and fit best in the architecture.

The study team was also given descriptions of the interactions of the tactics and common architecture patterns (taken from [16]). They could use this additional information to find the tactics that best fit the architecture

Each team was asked to create an architecture diagram that showed the architecture that incorporated all the tactics they used. The diagram should show all the necessary components and connectors.

After the exercise, we analyzed the resulting architectures for the appropriateness of the design. For a given task, we checked to see whether the fault tolerance requirement was satisfied, whether the tactics used were correctly designed, whether the teams had selected the best tactics, and whether they showed understanding of the impact of their decisions on the architecture, as measured by the accuracy of their effort estimates. For example, if the Ping-Echo tactic was selected in

the first task, the architecture must show a controlling component (preferably in the existing Broker component), and a communication link to each component, including those in the Pipes and Filters part of the architecture.

We analyzed the results and compared our analyses for agreement.

5. RESULTS

We compared the results of the two teams We begin by discussing the overall design data. For each task, we analyzed the tactics selected and how the tactics were inserted into the architecture.

Task 1 (Recommended tactic: Ping-Echo or Heartbeat)

The study team used both Heartbeat and Exceptions. There may be a bit of overlap between the two (see [2]), but they can be complementary. The Heartbeat was controlled by the existing monitor component. In addition, they replicated the monitor component as well as the Broker component, using Active Redundancy. They included the necessary connectors for the Heartbeat, as well as for transmitting the Exceptions. The use of active replication increases the likelihood of non-stop processing (availability), and is recommended.

The control team used both Heartbeat and Ping-Echo, as well as Exceptions. This is redundant. They created a monitor component, but did not make it part of the existing Broker component. Their connections to each component were not explicit. They also used Voting, apparently to arbitrate between results from the Heartbeat and the Ping-Echo. This is not a valid use of Voting, and in fact, wouldn't even work, since Voting requires three or more voters. In addition, the design of the Voting tactic was not clear in the architecture.

Task 2 (Recommended tactic: Spare)

Both the study and control teams used Spare, and used the Broker component to manage it. This is a very good fit.

Task 3 (Recommended tactic: Transaction Rollback)

We note that implementing Transaction Rollbacks in a Pipes and Filters architecture is particularly difficult and troublesome.

The study team used the Exceptions, which they already had from Task 1. They considered Transaction Rollback, but rejected it.

The control team also used Exceptions, which they added at this time. However, they did not fully add the necessary connections to make it work; it wasn't clear what component was responsible for handling the Exceptions. So although they used the same tactic as the study team, their solution was incomplete.

Task 4 (Recommended tactics: Passive Redundancy and optionally Process Monitor)

The study team used Passive Redundancy. Instead of using a Process Monitor to manage it, they connected it directly to the active component. In this scenario, the active component is capable of assessing its own health, so it could notify the redundant component. This is good and elegant.

The control team used an unnamed form of redundancy. The biggest problem, though, was that the redundant component was not connected to anything. Since the connections are the key difficulties, their proposed architecture portends problems when they actually implement this feature.

Table 3 summarizes the tactics used by each team for each task. It includes notes about connections.

Table 3. Tactic Use Summary

Task	Suggested Tactics	Study Team	Control Team
1	Ping-Echo or Heartbeat	Heartbeat, Exceptions, and Active Redundancy (connections good)	Heartbeat, Ping-Echo, Exceptions, and Voting (connections unclear)
2	Spare	Spare, good connection	Spare, good connection
3	Transaction Rollback	Exceptions (already in use), considered Transaction Rollback	Exceptions (connections and functionality missing)
4	Passive Re-dundancy, Process monitor (optional)	Passive Redundancy (direct connection; process monitor not needed)	Passive Redundancy (connections unknown)

We follow this in the next sections with a discussion of the data in light of the four research questions posed.

5.1 Satisfaction of FT Requirements

The question is whether the tactics selected satisfy the fault tolerance requirements of each task. We can summarize the results as shown in Table 4.

Table 4. Satisfaction of FT Requirements

Task	Study Team	Control Team
1	Yes	Yes
2	Yes	Yes
3	No, use of exceptions does not easily meet the need to roll back operations.	No, use of exceptions does not easily meet the need to roll back operations.
4	Yes	Probably: however, lack of connection information made it unclear whether the requirements were completely fulfilled.

Here we see that both teams were nearly identical in meeting the fault tolerance requirements introduced in each task. In each case except Task 3 they selected one or more tasks that fulfilled

the requirements. In Task 4, the control team did not show how the passively redundant component would be triggered to take control, so there is possibly a deficiency in meeting the requirement. However, the main problem is in design correctness, dealt with in the next question.

5.2 Design Correctness

The question concerns whether the tactics are designed correctly within the context of the architecture patterns used and there are no missing components or connectors. Strength in this area indicates an understanding of how a tactic must be implemented in the patterns used in the system. The following table summarizes the correctness of each task by each team.

Table 5. Correctness of Tactic Design

Task	Study Team	Control Team
1	Correct	Connectors for all tactics missing, and Voting design is wrong.
2	Correct	Correct
3	Correct	Missing connectors for Exceptions
4	Correct	Connection to redundant component entirely missing

Here we see that the study team generally did better than the control team. Task 1 was particularly problematic for the control team, and Task 4 had significant omissions.

5.3 Optimal Tactic Selection

Here we examine whether the tactics selected were the best choices, given the architecture patterns used. Strength here indicates an understanding of the impact of the tactics on the patterns, and tradeoffs of tactics.

Table 6. Optimal Tactic Selection

Task	Study Team	Control Team
1	Heartbeat is appropriate; Exception is probably not needed	Use of Heartbeat, Ping-Echo and Exceptions is unnecessary. Use of Voting is not called for. Solution far too complex.
2	Yes	Yes
3	If Exceptions really met the requirement, it would be a good alternative to rollback	If Exceptions really met the requirement, it would be a good alternative to rollback
4	Yes, in fact superior to suggested solution.	Missed needed connections; indicates lack of understanding of tactic's implementation in the patterns.

We see in table 6 that the study team was slightly better in Task 4, and significantly better in Task 1.

5.4 Understanding of Effort

This question relates to the participants' understanding of how the tactics are to be implemented in the patterns used in the architecture. For this we asked the two groups to estimate the difficulty of implementing the task, and compared it against evaluators' consensus of the difficulty. The groups scored the difficulty on a scale of 1 to 5, with 1 being the easiest. We then evaluated the difficulty of implementing the tactics that each group selected for each task, and rate them on the same scale of 1 to 5. A small difference between a team's estimate and the evaluators' estimate indicates a good understanding of what is required to implement the tactics in this architecture.

The difficulty estimates for each team are shown in Table 7.

Table 7. Difficulty Estimates, Study Team

Task	Evaluators' Estimate	Study Team's Estimate	Difference
1	4	5	+1
2	1	5	+4
3	1	1	0
4	1	2	+1
Average Magnitude			1.5

We see that except for Task 2, the control teams's estimates were close to the evaluators' estimates, although they were consistently higher.

Table 8 shows the difficulty estimates for the control team.

Table 8. Difficulty Estimates, Control Team

Task	Evaluators' Estimate	Control Team's Estimate	Difference
1	5	3	-2
2	1	3	+2
3	2	4	+2
4	1	2	+1
Average Magnitude			1.75

The estimate of 5 for task 1 may actually be an understatement; the proposed solution here would be very difficult and troublesome to implement correctly. We see a somewhat larger average difference than for the study team. This indicates they understood the impact of implementing the tactics somewhat better than the control team.

In all four questions, the study team did as well or better than the control team did.

5.5 Limitations

We note that there are several limitations that either threaten the validity of the study or limit its applicability. They are as follows:

1. The sample size was too small to draw any statistically significant conclusions. There were two groups. An

alternative would have been to have each person work individually; however, software architecture of non-trivial systems is generally a team effort. Individual architecture work would have thus been less realistic. As a result, we chose group assignments, and state our conclusions as observations rather than statistically established trends.

2. Analysis of the data was performed by ourselves, and not validated by independent researchers. Therefore, there may be inadvertent bias in favor of a positive result.

3. The participants had limited experience with the design of fault-tolerant systems. In fact, we observed that the lack of fault tolerance experience behavior had some impact on designs (noted in section 5). We therefore limit our recommendations to those with limited experience in fault tolerance.

6. INTERPRETATION OF RESULTS

In the designs, the first question is whether the tactics selected meet the needs of the fault tolerance added in the task. We see that for the most part, both groups selected tactics that would satisfy the task. The one exception was Task 3, where both groups used Exceptions rather than Transaction Rollback. This may be because the system did not have natural transactions, and therefore, Transaction Rollback did not seem like a good fit. We note that the study team did consider it and rejected it. The fact that both groups selected the same tactic supports the notion that it was an artifact of the problem presented. Overall, the success by both groups indicates that they understood how the tactics would implement fault tolerance. This was notable in that they had little fault tolerant design experience.

The second question is how well the tactics selected met the architecture. Tactics used by both groups necessarily have the same impact on the architecture, so we need examine only the tactics uniquely selected by each team. The study team selected Active Redundancy for Task 1. This is an acceptable match for the architecture (and the group included it appropriately in the architecture).The control team selected Voting. Voting can be a good match for the Pipes and Filters architecture, but it was not used correctly. In addition, it was not even a good fit for the application under design. Its use demonstrated a lack of understanding of how it would be implemented in the architecture patterns present, and its impact on those patterns. It appears that the control team suffered from lack of knowledge of the tactic-pattern interaction that the study team was given. This supports the hypothesis that such knowledge helps architects come up with a correct design.

A related question is how well the architectures accounted for the implementation of the tactics selected. In particular, how are components modified or duplicated, are additional components added, and are connectors between components either modified or added as needed? Architecture diagrams generally do not show changes in behavior within components or connectors, but readily show new and replicated components, as well as new connectors. Incorrect or missing components and connectors in an architecture diagram are an indicator of trouble – implementation is likely to be more troublesome than anticipated. It may indicate that the architects did not consider or even understand the components and their interactions.

In our study, the architecture produced by the study team consistently showed all the components and connectors needed to implement the tactics they selected. In contrast, the architecture produced by the control team was missing several connectors that were needed to implement their selected tactics. This is a significant omission, as the connections require changes to the all components involved in the connection. Such changes can involve timing (such as ping-echo responses) or synchronization (such as responses to exceptions), thus the changes may be significant. It appears that the tactic-pattern interaction information may have helped the study team consider the required connections more thoroughly. This tends to support the hypothesis that such knowledge helps architects come up with a correct design.

One additional observation concerning the goodness of design concerns "false paths" – design alternatives that were considered but rejected. Jansen et al. consider this an important part of architectural knowledge – it is part of the rationale for making architectural decisions [19]. In our study, we attempted to quantify the number of alternatives considered and rejected, however we slipped up and did not measure it consistently. We did observe, however, that the study team considered and rejected numerous alternatives (such as their consideration of transaction rollback in Task 3). This leaves an open question of whether tactic-pattern interaction information can help architects by helping them consider (and reject) many alternatives. This is probably best understood through studies that include interviews of the subjects, to learn how alternatives were considered.

One additional observation was notable: Both groups tended to over-engineer the solutions. They tended to use more tactics than was really necessary. (For example, the control group used both Ping-Echo and Heartbeat in the first task.) There are at least two possible motivations for this behavior. First, the experimental setting encouraged the participants to do the best job they could; therefore, when in doubt, they added more tactics. We saw evidence of this attitude in each group: the study team added redundancy to a component that wasn't really needed in the first task. And the control team added the Voting tactic more or less gratuitously. We surmise that this motivation is significantly less in real project settings.

The second possible reason is that virtually all the participants had little experience designing fault-tolerant software, and the tactics were somewhat new to them. Therefore, they had a tendency to use as many as they could. We have observed a similar phenomenon with respect to software design patterns before – when people are first introduced to design patterns [9], they tend to overuse them, until they gain more experience with the patterns. Similarly, as people gain fault tolerance experience, they might avoid overuse of tactics.

7. Related Work

Various methods of architectural design (synthesis) have been proposed; some of the most prominent methods have been summarized by Hofmeister et al [17]. All these methods consider important quality attributes including fault tolerance. The Attribute-Driven Design method specifically focuses on using important quality attributes as drivers for the architecture [2]. The information of how fault tolerance tactics and

architecture patterns interact can be used in any of these architecture design methods to help architects make informed decisions. This study begins to explore how this can be done.

There are also several methods of architectural evaluation; such as ATAM [6]. Most architectural evaluation methods consider quality attributes, and the tactic-architecture interaction information can also be useful to identify potential architectural issues. This study does not concern itself specifically with architectural evaluation, but may begin to establish the validity of such data.

Architectures can be considered to be a set of design decisions [3]. Tactic-pattern interaction information can enhance the rationale information behind such decisions.

Hanmer has written a collection of patterns for developing fault-tolerant software [12]. These are not architectural patterns, such as are described above, but are more like the tactics described here. Similarly, Utas describes various methods of achieving fault tolerance [24]. These are analogous to the fault tolerance tactics described here, and in fact, several are the same. The measures described in these works also have architectural implications. This work provides a model for understanding the impact of these measures on architecture patterns.

In [16], we described the nature of the impact of fault tolerance tactics on the architecture patterns in a system. We categorized the types of impact, (e.g., modification of existing components, replication of components, adding new components) and propose a scale of impact of implementing tactics on the architecture patterns. We suggested such information might be useful to architects; this study begins to investigate how it can be useful.

In the past we described a taxonomy of nearly all of the well-known architecture patterns [1]. In it we show several patterns that can be used as alternatives to each other. This information, coupled with information about the impact of implementing various tactics in these patterns, can potentially provide architects with information about the benefits and liabilities of using alternate architecture patterns. This study can help establish its usefulness.

8. CONCLUSIONS AND FUTURE WORK

This limited exploratory study suggests several issues that are worth further study.

The study suggests that the design of tactics can impact the architectural design of a system, and perturb the architecture patterns used therein. It appears that understanding of how they interact is useful and important in architectural design. We observed that architecture teams with this information appear to be somewhat better at designing fault tolerance measures in a system than those without the information. In particular, they can better select tactics that are compatible with the architecture. They also better understand the impact of the tactics in terms of the structure as well as the expected difficulty of implementation. Since this was a limited study of a small set of people who have architectural experience, but have limited fault tolerance experience, we recommend that further study be done with individuals with more fault tolerance experience. It may be particularly useful to study those who have fault tolerance experience, but less architecture experience, to determine how helpful this information is to neophyte architects.

We had positive results with a small set of patterns and fault tolerance tactics. We propose that studies be undertaken with large sets of fault tolerance measures (for example, [12] and [24]), and patterns (for example, [5] and [1]). Ideally, this should become a library of architecture patterns and fault tolerance tactics, and how they interact. This could be a useful reference for architects of fault tolerant software.

9. REFERENCES

[1] Avgeriou, P. and Zdun, U. Architectural Patterns Revisited – a Pattern Language. In Proc. Of *10th European Conference on Pattern Languages of Programs (EuroPLoP 2005)*, (Irsee, Germany, July 6-10, 2005).

[2] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice,2nd ed*. Addison-Wesley, Reading, MA, 2003.

[3] Bosch, J.: Software architecture: The next step" Software Architecture, First European Workshop (EWSA), volume 3047 of LNCS, Springer (May 2004) 194–199.

[4] Buschmann F. et al., *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, Chichester, England, 1996.

[5] Buschmann F., Henney, K. and Schmidt, D, *Pattern-Oriented Software Architecture volume 4: A Pattern Language for Distributed Computing*. Wiley, 2007.

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

[7] de Lemos,R., Asterio de Castro Guerra, R., and Rubira, C. M. A Fault-Tolerant Architectural Approach for Dependable Systems, in *IEEE Software,* 23,2, March/April 2006, 80-87.

[8] Ferreira, G. R., Rubira, C. M., and Lemos, R. d. 2001. Explicit Representation of Exception Handling in the Development of Dependable Component-Based Systems. In *the 6th IEEE international Symposium on High-Assurance Systems Engineering: Special Topic: Impact of Networking* (October 24 - 26, 2001). HASE. IEEE Computer Society, Washington, DC, 182-193.

[9] Gamma, E., et al.: 'Design Patterns: Elements of Reusable Object-Oriented Software' (Addison-Wesley, 1995)

[10] Garcia, A. F., and Rubira, C. M. An Archietctural-based Reflective Approach to Incorporating Exception Handling into Dependable Software. In *Advances in Exception Handling Techniques*, Springer-Verlag, LNCS-2022, 2001, 189-206.

[11] Garcia, A. F., Rubira, C. M. F., Romanovsky, A. B., and Xu, J. A., A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software, in *Journal of Systems and Software* 59, 2, 2001, 197-222.

[12] Hanmer, R. *Patterns for Fault Tolerant Software,* Wiley, Chichester, England, 2007.

[13] Harrison, N., and Avgeriou, P. 2008. Analysis of Architecture Pattern Usage in Legacy System Architecture Documentation. In *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008) - Volume 00* (February 18 - 21, 2008). WICSA. IEEE Computer Society, Washington, DC, 147-156.

[14] Harrison, N., and Avgeriou, P. Leveraging Architecture Patterns to Satisfy Quality Attributes, In proc. *First European Conference on Software Architecture*, Madrid, Sept 24-26, 2007, Springer LNCS.

[15] Harrison, N., and Avgeriou, P. 2007. Pattern-Driven Architectural Partitioning: Balancing Functional and Non-functional Requirements. In *Proceedings of the Second international Conference on Digital Telecommunications* (July 01 - 05, 2007). ICDT. IEEE Computer Society, Washington, DC, 21.

[16] 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.

[17] Hofmeister, C.;,Kruchten, P., Nord, R.L.; Obbink, H., Ran, A. & America, P. Generalizing a Model of Software Architecture Design from Five Industrial Approaches, In *Journal of Systems and Software*, 30,1, Elsevier, 2007, 106-126.

[18] Issarny, V. and Banâtre J. 2001. Architecture-based Exception Handling. In *Proceedings of the 34th Annual Hawaii international Conference on System Sciences (Hicss-34)-Volume 9 - Volume 9* (January 03 - 06, 2001).

[19] Jansen, A., Bosch, J., and Avgeriou, P. 2008. Documenting after the fact: Recovering architectural design decisions. *J. Syst. Softw.* 81, 4 (Apr. 2008), 536-557.

[20] Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., Emam, K. E., and Rosenberg, J. 2002. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.* 28, 8 (Aug. 2002), 721-734.

[21] Laibinis, L. and Troubitsyna, E. 2004. Fault Tolerance in a Layered Architecture: A General Specification Pattern in B. In *Proceedings of the Software Engineering and Formal Methods, Second international Conference* (September 28 - 30, 2004). SEFM. IEEE Computer Society, Washington, DC, 346-355.

[22] Muccini, H., Pelliccione, P., and Romanovsky, A. 2007. Architecting Fault Tolerant Systems. In *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture* (January 06 - 09, 2007). WICSA. IEEE Computer Society, Washington, DC, 43.

[23] Shaw, M. and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline.* Addison-Wesley, Reading, MA, 1996.

[24] Utas, G. Robust communications Software: Extreme Availability, Reliability and Scalability for Carrier-Grade Systems, Wiley, Chichester, England , 2005.

[25] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. *Experimentation in Software Engineering: an Introduction*. Kluwer 2000.

10. APPENDIX

The following is the tactic data for the Pipes and Filters and Borker patterns.

Pipes and Filters

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 don't 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: - -: Checkpointing 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: ~: The first filter might create the transactions. However, filters work more naturally on streaming data than on transaction-oriented data.

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.)

Broker

1. Fault Detection

a. Ping/Echo: + +: The Broker component can implement a ping/echo and can even have a message serve double duty as a request and as a ping message. This is both efficient and easy to implement. (Implemented in the Pattern)

b. Heartbeat: +: As in ping/echo, the Broker is the monitor component. The other components must implement a heartbeat mechanism, with messages independent of the normal control messages. (Minor modifications to components needed.)

2. Recovery – Preparation and Repair

a. Voting: + +: The broker distributes work to multiple servers, acts as the arbiter among them. (Implemented in the pattern. Different voting components are implemented as server components.)

b. Active Redundancy: + +: The broker component provides a natural mechanism for distributing the same messages to redundant servers, and management of the swap to the redundant server. Servers might be replicated, or that might be done already. (Implemented in the Pattern.)

c. Passive Redundancy: +: The Broker component manages updating the spare with the active's state. Active Redundancy is such a good fit that it is recommended over this tactic. However, it is a good fit for duplicating the Broker component. (Minor modifications to the active and backup components.)

d. Spare: + +: If the system has multiple servers with different responsibilities, then the broker component manages bringing in a spare for any of the failed servers. (Implemented in the pattern; a few minor modifications may be needed.)

3. Recovery – Reintroduction

a. Shadow: + +: The broker component is a natural place for monitoring servers that return to service. The broker can keep track of the health of the servers, and mark a server as a shadow until it returns to full operation. (Implemented in the pattern)

b. State Resynchronization: + +: When a server comes back into service, the Broker component can send it state information to synchronize it. (Implemented in the pattern.)

c. Rollback: + +: Broker-controlled systems tend to be transaction-oriented. The actual rollback happens within the server components, while the control of rollback may be within the server, or in the Broker, depending on the application. (Implemented in the pattern.)

4. Prevention

a. Removal from Service: + +: The broker component can act as an arbiter for the server, and remove it from service if it detects error conditions. (Implemented in the pattern.)

b. Transaction: + +: Broker systems tend naturally to be transaction-oriented. The units of transaction would be defined in terms of work done on the server. (Implemented in the pattern.)

c. Process Monitor: + +: It is natural for the broker component to detect if a server fails and restart it. (Implemented in the pattern.)