1 Introduction

1.1 Improving the World through Software

One of the most dramatic innovations in the history of the world is the stored program computer and the software that runs on it. Few innovations have had such a large impact in such a short time. In just a few decades, software has gone from small programs that performed specialized scientific calculations to extremely complex system that affect nearly every aspect of our lives. Indeed, software is pervasive; we find software not only in home and corporate computers, but in virtually every business, in cars, mobile phones, and even in simple home appliances such as toasters.

Software improves our lives in many ways. These are only a few examples:

- Software makes many tasks easy to perform. For example, one can purchase almost anything online, without the need to go to a store.

- Software manages systems where a high degree of reliability is needed. For example, software ensures that financial transactions are correct – that your paycheck is deposited in your bank account, and not somebody else’s!

- Software performs tasks quickly where speed is of the essence. Software that controls airplane flight responds quickly to atmospheric changes, helping to maintain level flight.

A consequence of the increasing capabilities of software is that software programs have become extremely complex. Indeed, software systems are generally considered to be the most complex entities ever created by humans. This is not a new phenomenon; the challenges of increasingly complex software have long been recognized. In 1968, a NATO conference was convened to address this topic. The conference coined the term “software engineering” as an ideal approach to designing and developing highly complex software systems. Efforts to improve the state of software engineering continue to this day.

A key way to deal with highly complex software applications is to organize the software in modules that can be understood and implemented partially independently of each other (see [106, 107].) The systematic organization of software gave rise to the discipline of software architecture [108, 121]. Software architecture has emerged as an important discipline in the creation of complex software systems.
1.2 Software Architecture

In order to understand software architecture, let us begin with architecture of buildings. One can usually build shed with four walls, a door, and a roof with little more than plans kept in one’s head. However, this is impossible with more complex buildings. Even small homes require an overall architecture and plans that provide the overall layout of the house and guide the workers through its construction.

This is also true of software: even moderately complex software applications require an overall plan that lays out the modular structure of the system and guides its implementation – they require a software architecture. Like buildings, software has an overall structure that governs its construction and evolution. But of course, software is more than structure: software runs. In fact, a complete picture of software must include not only its static structure but its runtime behavior. Nevertheless, the comparison of the architecture of buildings to the architecture of software is useful. Indeed, the three principles of architecture put forth by Vitruvius over 2000 years ago – firmness, utility, and beauty – are qualities that software should aspire to even today.

As software systems became larger and more complex, the need for systematic large-scale organization of the structure and behavior became apparent; thus the discipline of software architecture was born [108, 121]. While there are several definitions of “software architecture”, one popular definition is as follows:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [15].

The architecture of a system can have a significant impact on the characteristics of the system. The architecture of a building affects the building’s characteristics in important ways. The following are just two examples:

- The usability of the building is affected by its layout. The placement of rooms and doors between them affect the traffic flow, and can make rooms more or less pleasant to be in. The placement of windows affect room’s ambiance [4].

- The architecture specifies structural elements that affect the reliability and strength of the building. Flying buttresses were incorporated into Gothic cathedrals in order to strengthen the walls as they ascended higher and higher.

This is also true of software. The architecture of a software system affects the externally visible characteristics of the software. For example:
The author was once involved in the test of a large system. The architecture of the system was designed so that four simultaneous transactions were allowed. Further architectural decisions resulted in database queries and updates that were very expensive. When this was combined with the overhead of handling simultaneous transactions, the result was interminably long transaction delays, and great user dissatisfaction. The architecture had a strong impact on performance, and through it, on usability.

Fault-tolerant applications require that the system recover from external error conditions; for example, an online banking system must correctly abort a partially completed money transfer if the communication link drops before the transfer is completed. The transfer may fail, but a partial transfer, such as money taken from one account, but not deposited to the other, must never happen! The software architecture likely specifies how (and where in the software) the faults are handled, which may make correct recovery easy or difficult.

From these examples we see that architecture impacts systems in user-visible ways, in that it constrains and shapes the implementation. These “user-visible ways” are often the quality attributes of the system. Unfortunately, here is an essential dilemma of architecture. The architecture must be established early in the design of the system, and the architecture affects certain important characteristics of the system. However, one cannot fully validate whether the characteristics of the system meet the system’s quality attribute requirements until the entire system has been constructed – far too late to easily change the architecture if needed. This means that architecture is in some way a foray into the unknown. This is true whether the systems be buildings or software. Let us explore examples in buildings:

- The builders of a Gothic cathedral could not know for sure if a cathedral would fall down until after the roof had been completed. Only then would the strength of the architecture be fully validated. One of the most famous examples concerns a ship; the Swedish warship Vasa. On its maiden voyage, in full view of many Swedish nobility, it tipped over and sank in the Stockholm harbor. It was fundamentally unstable.

In software, the issue is the same. Clements, et al. state, “Modifiability, performance, security, availability, reliability – all of these are precast once the architecture is laid down. No amount of tuning or clever implementation tricks will wring any of these qualities out of a poorly architected system.” [38] Yet the system’s performance with respect to these characteristics cannot be validated until the after the system is implemented. For example:
- The performance of real-time-critical systems cannot be fully known until the entire system is complete. After all, any piece of the system might conceivably take too much real-time processing, degrading the entire system’s performance.

- High reliability is a system-wide characteristic: the reliability of a software system is only as strong as its weakest link. Therefore one cannot validate the reliability of the system until all the links are in place; until all the code has been completed and put together.

- An architectural design principle is that one should partition (e.g. in different modules) the system according to the expected changes, so as to localize the impact of future changes. The author worked on a system where this principle was ignored; as a result, any substantial enhancement caused nearly every significant module to change. It appeared that the original architects were unaware that their architecture would be difficult to enhance, as no architecture documentation even mentioned enhancability. Unfortunately, this did not become apparent until the product had been released and we were working on subsequent releases.

Fortunately, we are not left wholly to our own devices. We can build on previous architects’ expertise. We can follow the architectural patterns as we create buildings or software. These architectural patterns are essential tools for every journeyman architect of software and buildings alike. We can use these patterns as well as other knowledge of previously constructed systems to evaluate architectures before systems are fully constructed.

The most striking pattern in building architecture may be the flying buttress. This pattern allowed architects to create walls that were higher and thinner than before. One can see the effect of flying buttresses in the soaring naves and towering stained glass of the Gothic cathedrals of Europe. It may be the single architectural innovation that advanced the Romanesque architectural period to the Gothic period.

But software is a new discipline, and does not have a history of hundreds of years from which to draw such patterns. Nevertheless, patterns of software architecture have begun to emerge. These patterns are guides to the high level structure and behavior of software systems: they show architects how it has been done before. Architects are increasingly standing on the shoulders of their predecessors – using established patterns of software architecture.

The especially good news is that by following architecture patterns, one can design systems that will have the desired qualities, because the patterns impart those qualities. The flying buttresses gave strength to the walls, and the architects could
rely on them even before the walls were built. In software, patterns such as Broker provide a platform for highly reliable systems, and Pipes and Filters helps create a system that is really fast. Thus software architecture patterns hold great promise for designing the systems of the future.

Of course, since software architecture is a recent development in an immature domain, the understanding of how software architecture patterns help architects design systems is still in its infancy. Yet it holds great promise: it can potentially help architects design systems better, and can even help detect architectural weak spots – before the architecture is fully cast in stone. These potential benefits form the core of the research behind this thesis.

1.3 Key Concepts and Terms
Before exploring the key problem and research questions, I must give some key definitions of terms related to software and software architecture that were used in this research.

1.3.1 Architecture Patterns
Patterns are solutions to recurring problems. A software pattern describes a problem and the context of the problem, and an associated generic solution to the problem. The best known software patterns describe solutions to object-oriented design problems [52], but patterns have been used in many aspects of software design, coding, and development. Architecture patterns describe an abstract, high-level system structure and its associated behavior. [33]. Architecture patterns generally dictate a particular high-level, modular system decomposition. Numerous patterns have been written for software architecture, and can be used in various software architecture methods [15, 28, 81, 121].

An important advantage of patterns is that the consequences of using the architecture pattern are part of the pattern. The result of applying a pattern is usually documented as “consequences” or “resulting context” and is generally labeled as positive (“benefits”) or negative (“liabilities”). Many of the benefits and liabilities concern quality attributes; for example, a benefit of the Layers pattern states, “Standardized interfaces between layers usually confine the effect of code changes to the layer that is changed … This supports the portability of a system.” A liability of the same pattern states: “A layered architecture is usually less efficient than, say, a monolithic structure of a ‘sea of objects’.” [33] However, the information is incomplete; it does not cover all the well-known quality attributes. The architecture patterns almost always highlight only a few quality attributes, perhaps because pattern writers tend to favor conciseness over comprehensiveness.
In addition, this information is currently inadequate. The impact of a pattern on a quality attribute is often given simply as a positive or negative impact [39, 62], but how that maps to a real system is not explained. In addition, architecture patterns are general solutions, thus their impact on quality attributes largely depends on how the pattern is used in the architectural design. Even detailed descriptions of the impact, such as given in the excerpt above, currently do not explain how impact on quality attributes may be mitigated, or how the architecture can be changed to achieve the quality attribute of interest. For example, the excerpt above does not give possible variations to the Layers pattern that could reduce the performance impact.

1.3.2 Quality Attributes
Quality attributes are characteristics that the system has, as opposed to what the system does [124], such as usability, maintainability, performance, and reliability. Quality attributes are not simply met or not met, but rather, satisfaction is along a scale, generally viewed within the specific context of a scenario: given a system state and specific input, the output is required to extend within specific limits [12, 15]. A particular challenge of quality attributes is that because they tend to be system-wide characteristics, system-wide approaches are needed to satisfy them; these approaches are defined at the system architecture level and not the component level.

1.3.3 Tactics
Tactics are measures taken to improve specific problems within quality attributes [15]. For example, a duplication scheme to improve the nonstop operation of the system (a subset of reliability) is a tactic. Unlike architecture patterns, tactics do not intend to shape the overall structure of the system. Tactics may be “design time,” or overall approaches to design and implementation, such as “hide information” to improve modifiability, or may be “runtime tactics,” which are features directed at a particular aspect of a quality attribute, such as “authenticate users” to improve security. In this thesis we concern ourselves only with runtime tactics.

Because tactics present specific approaches to improving quality attributes, they are similar to the general concept of patterns. Indeed, some tactics have been written as patterns (see [126] and [61].) They are analogous to design patterns [52] in that that they directly support implementation of a feature within a system. On the other hand, architecture patterns are system-wide; they describe “major architectural abstractions” [116]. As such, design patterns and tactics are used in implementation, but within the framework of the architectural abstractions created by the architecture patterns.
1.3.4 Architecture Patterns and Tactics

Tactics have structure and behavior, and as such, they can impact architecture patterns in various ways. In some cases, a tactic may be easily implemented using the same structures (and compatible behavior) as a particular architecture pattern. On the other hand, a tactic may require significant changes to structure and behavior of the pattern, or may require entirely new structures and behavior. In such a case, implementation of the tactic, as well as future maintenance of the system are considerably more difficult and error-prone.

Because of the importance of quality attributes, it is critical that they be considered during early design; during system architectural design. Indeed, we find that architects often consider them simultaneously [63, 69]. However, if architects do not understand the relationship between the architecture patterns and the tactics being used, they risk making architectural decisions (either concerning which patterns to use or which tactics to use) that can make the software difficult to implement correctly and maintain.

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

1.3.5 The Architectural Design Process

The design of software architecture takes place early in the software development cycle, in order to provide a framework in which detailed software design and implementation may take place. A general model of the software architectural design process, proposed by Hofmeister, et al. [72] consists of the following activities:

- **Architectural Analysis**, which articulates architecturally significant requirements based on the architectural concerns and context of the desired system
- **Architectural Synthesis**, which produces candidate architectures that address the requirements from architectural analysis
- **Architectural Evaluation**, which ensures that the architectural decisions are the right ones (i.e., evaluates and selects among candidate architectures.)

These activities are performed iteratively, as architectural concerns, context, and requirements become increasingly clear.
In order to understand how to improve the satisfaction of quality attributes at time or architectural design, it is important to understand the architectural design process at a high level. The terms Architectural Analysis, Architecture Synthesis, and Architectural Evaluation will be used in the problem statement and throughout the thesis.

1.4 Problem Statement
During the process of software architectural design, architects select architecture patterns to be used. It is easy to see the structure of the patterns, but their impact on the all-important quality attributes is not apparent from observing the patterns. That means that one might select patterns that are incompatible with or even detrimental to the satisfaction of quality attributes. However, one cannot easily ascertain whether a quality attribute is sufficiently satisfied until the system is largely complete. By this time, architectural changes can cause significant disruption to the code already written.

♦ ♦ ♦ ♦

Architects must select architecture patterns early, yet the impact of these decisions on the quality attributes is not fully understood until it is too late to easily change the architecture.

Characteristics of architectural design that contribute to this include the following:

- While the structure of the patterns is visible, their impact on quality attributes is not apparent from their structure.
- Quality attributes are system-wide characteristics, and therefore their satisfaction cannot be fully ascertained until the system is largely complete.
- A system typically has multiple important quality attributes, thus the interaction of patterns and quality attributes is complex.
- Systems generally use multiple patterns in their architecture, which complicates the interaction of patterns with quality attributes even more.

The consequence of this problem is that software may fail to meet its quality attribute requirements. The consequences can be serious: at the very least, users may experience inconvenience and frustration. Beyond inconvenience, software quality attribute failures may burden users and software providers with financial liabilities. Even injuries and deaths have been attributed to software quality attribute failures.
In this thesis, I present techniques that help architects select patterns that improve the software architecture with respect to satisfaction of the system’s quality attributes. In order to come to these techniques, it was necessary to investigate several research questions.

1.5 Research Questions
The driving motivation for this thesis is to improve the quality of software systems, especially with respect to their quality attributes. There is a relationship between a system’s quality attributes and the architecture patterns used in its design. Therefore, the main research question is:

"How can architects leverage patterns to create architectures that meet quality attribute requirements during the phase of software architectural design?"

To begin with, let us examine the current state of the practice in software architecture. This gives us an idea of how architects leverage architecture patterns in the design of software architectures. We are particularly interested in how extensively architecture patterns are used, as this will give us an idea of the potential impact of architecture patterns. We also want to know whether different patterns are used in different application domains, as that may indicate patterns that are suited to particular domains (or the quality attributes that are prominent in those domains.) This brings us to this question:

*RQ-1: How common are architecture patterns in software architectures? In particular, which patterns are commonly found individually and in pairs in certain application domains?*

Understanding how commonly architecture patterns are used in software architectures gives us a foundation upon which we can delve further: depending on the frequency and other characteristics of their application, it is important to understand how their use impacts important quality attributes of systems. The application of a pattern constitutes a major architecture decision that achieves the satisfaction of a quality attribute or the lack thereof.

Architectural decisions are important artifacts of architectural knowledge, as they help future developers understand the architectural structure and the reasons particular structures were originally used [see 81]. The next research question explores the types of decisions that are embodied in architecture patterns.

*RQ-2: What is the relationship between architecture patterns and architectural decisions, particularly those concerned with quality attributes?*
An extremely important type of rationale of architectural decisions concerns quality attributes: architects make decisions to apply patterns mostly because of their consequences on the quality attributes. It stands to reason that certain patterns and quality attributes are compatible with each other, while others are less so.

Architects should select patterns that are good fits for the important quality attributes of the system under design. However, tactics are often employed to improve quality attributes, and the tactics may well be selected independently of the patterns. This may lead to conflicts for which tradeoffs must be made. Therefore, in order to support architects in making complex tradeoff decisions, we need to explore the relationship between patterns and tactics. To begin with, we expect that tactics themselves have impact on, and are impacted by the architecture patterns, and we wish to explore the nature of this mutual impact. The next question explores the types of impact they can have on each other.

*RQ-3: What model describes the interaction between patterns and the tactics one uses to implement quality attributes?*

With any model, it is important to understand how useful it is when applied to real-life situations. Therefore, we ask a related sub-question:

*RQ-3a: What do we learn about patterns and satisfying quality attributes through the application of this model?*

This model provides a fundamental understanding of the interaction of individual patterns and tactics, and we see that the model provides insights about how individual patterns are related to the satisfaction of quality attributes. However, nearly all software systems used in industry incorporate multiple architecture patterns and implement multiple tactics. Therefore, we need further insight into the relationship of multiple tactics and multiple patterns in a system. This brings us to the next research question.

*RQ-4: In a complex system requiring the use of multiple patterns and multiple tactics, what characteristics of the patterns, the tactics, and even the system itself influence where and how tactics are most effectively implemented?*

With an understanding of how patterns and tactics interact both individually and collectively, we are prepared to consider how patterns can be used to make the process of software architecting more effective. I first explore the use of patterns in the architectural analysis and synthesis activities, followed by exploring patterns in the architecture evaluation activities.
RQ-5: How does one incorporate patterns into the architectural analysis and synthesis phases of architectural design in order to help the architect consider how the structure impacts the satisfaction of quality attributes?

Patterns should also be useful in architectural evaluation activities. I consider whether patterns can be a useful tool in architecture reviews to uncover potential software implementation issues related to quality attributes. If so, patterns might be used as a way to augment architecture reviews.

RQ-6: How can one gain insight into the impact of the architecture patterns used on quality attributes early in the development cycle – while the architecture can still be readily modified?

1.5.1 Research Question Summary and Response

At this point, let us revisit the main research question, which is, “How can architects leverage patterns to create architectures that meet quality attribute requirements, during the analysis, synthesis, and evaluation phases of software architecture?” We begin to answer this question by doing background research into the use of architecture patterns.

From RQ-1 we learn that patterns are found in nearly all industrial systems, and that different application domains do have different patterns that are commonly used. Since they are ubiquitous, we then examined their use with respect to the formulation of architectures and particularly making architecture decisions.

From RQ-2 we learn that architecture patterns embody critical architectural decisions, particularly concerning quality attributes. The fact that they embody decisions concerning quality attributes is significant, and leads us to consider that some patterns might be used because of their impact on quality attributes. If so, then patterns may have natural affinity for certain quality attributes, and vice versa. Research confirms this, and reveals that certain patterns are particularly good fits for some quality attributes, but are not good fits for other quality attributes.

From RQ-3 we explore what lies behind this relationship between the patterns and quality attributes. As quality attributes are implemented using tactics, we attempt to form a model of how quality attributes and tactics are related. The resulting model shows that tactics used to improve quality attributes are implemented within the structures of the architecture patterns, and therefore, some patterns are naturally a better or worse fit for particular tactics. Of course, since most systems employ multiple patterns and use multiple tactics, the model must extend to such systems.

Research question RQ-3a showed that application of the model of pattern and tactic interaction reveals that the compatibility of a pattern with a quality attribute can
have a very wide range, from highly compatible to highly incompatible. The interaction between patterns and individual tactics provides reasons behind the level of compatibility. Architects can use this information to guide them in making architectural decisions.

The research of RQ-4 shows that in systems with multiple patterns, the natural compatibility of a pattern with a tactic, along with characteristics of the tactic itself and the quality requirements of the system under design combine to determine the optimal implementation place for a tactic.

We then consider how to take advantage of this model in order to help architects pick the best patterns. The process resulting from RQ-5 shows that this information is input into the architectural design process. In the analysis and synthesis phases, one uses an iterative process to select patterns for use, based on their impact on the important quality attributes of the system being designed.

The research of RQ-6 shows how in the evaluation phase, one can evaluate the architecture patterns in use to determine how readily the important quality attributes of the system will be fulfilled. One examines the natural fit of the patterns with the tactics associated with the quality attributes to help evaluate the appropriateness of the architectural design for the system being designed. This helps one make tradeoff decisions concerning which patterns and tactics to use.

In summary, one can use an understanding of the interaction of patterns and tactics, along with the system’s requirements, during system architectural design to both create an effective architecture and evaluate architecture for support of quality attributes. This can help software architects understand the impact of their architecture patterns on the system’s quality attributes, and learn this early in the system development cycle. Thus they can create architectures that meet the quality attribute requirements, and reduce the errors and costs that arise from systems that fail to meet quality attribute requirements.

1.6 Research Methods

1.6.1 Introduction
Because of the newness of the fields of computer science and software engineering, and the abstract nature of the fields, research methods are still somewhat ill-defined. Efforts are being made to define research methods, motivated in part by the working group on computer science research methods as part of the ACM special interest group on computer science education (SIGCSE-CSRM) [123]. A framework of a research process has been defined which describes four key questions concerning research, and the iterative process that links these questions [74]. This framework is illustrated in figure 1.1.
Glass, et al. described and classified the software engineering research methods found in published software engineering research studies [57]. These were further expanded and organized by Holz, et al. [74].

These research methods chiefly provide guidelines for researchers in formulating questions (box A in figure 1.1), collecting data (box B), analysis of the data (box C), and drawing conclusions based on that analysis (box D). In order to make this general framework useful to software engineering researchers, Shaw [117] proposes a research classification framework. She classifies research accordingly:

- **Research Setting**: This is a classification of research questions, based on what one wishes to learn; for example, the research setting “Feasibility” contains questions to learn whether solutions can be found; e.g., “Is it possible to accomplish X?” This addresses the general question stated in figure 1.1, “What do we want to achieve?”

- **Research Results**: Classification of the analysis of the data obtained through the research methods. This addresses the question, “What do we do with the data?”

- **Validation Techniques**: This classifies how to establish the validity of the research results. This addresses the question, “Have we achieved our goal?” as well as the sub-issues of drawing conclusions and identifying limitations.

Taken together, these works provide general classification guidelines for types of research questions to ask, how to collect and analyze data, and how to judge the
validity of the research conclusions. The following sections show how the research questions in this thesis fit within these classifications.

### 1.6.2 Research Methods Used

In their study of software engineering journal publications, Glass, et al. [57] found that much of the software engineering research is qualitative in nature. The main methods used were conceptual analysis and concept implementation. Holz, et al. provide more detailed descriptions of research methods, identifying 55 different research methods used in computer science and software engineering. In this research, the following four methods have been used:

- **Descriptive Research:** this method of research involves collecting data and forming theories or models of the phenomenon under study; “theories or models are developed and described to provide the input for developing units of the theory, its laws of interaction, system states, and model boundaries.” [99].

In this research, descriptive research is used to increase understanding of the nature of architecture patterns, quality attributes and tactics, and how they relate to each other. Specifically, descriptive research is used in the following research questions:

- **RQ-2:** *What is the relationship between architecture patterns and architectural decisions, particularly those concerned with quality attributes?* This research compares elements of architectural decisions with decisions associated with architecture patterns and showed that they correlate. It also identifies the natural fit of patterns and quality attributes by analysis of the patterns’ structure and behavior compared to the needs of the quality attributes. This research (and the answer to this question) is found in chapter 3.

- **RQ-3:** *What model describes the interaction between patterns and the tactics one uses to implement quality attributes?* This research forms a model of this interaction, described in chapter 4.

- **RQ-4:** *In a complex system requiring the use of multiple patterns and multiple tactics, what characteristics of the patterns, the tactics, and even the system itself influence where and how tactics are most effectively implemented?* This research extends the model formed as a result of RQ-3, and is found in chapter 6.
• **Developmental Research:** this method of research goes beyond descriptive research in order to propose solutions: “... generating knowledge for explaining or solving general problems” [99].

Developmental research is used in this research to develop methods of effectively using architecture patterns to assist in creating and evaluating architectures that satisfy quality attributes. This is addressed in the following research questions:

- **RQ-5:** *How does one incorporate patterns into the architectural analysis and synthesis phases of architectural design in order to help the architect consider how the structure impacts the satisfaction of quality attributes?* This research developed a method for using architecture patterns in the process software architecture analysis and synthesis, which is described in chapter 7.

- **RQ-6:** *How can one gain insight into the impact of the architecture patterns used on quality attributes early in the development cycle – while the architecture can still be readily modified?* This research developed a method for using architecture patterns in the software architecture evaluation process; see chapter 8.

• **Static Analysis:** this research method is a special type of Historical Method, which involves collection of data from completed projects. The Static Analysis method performs analysis of the product as follows: “... analyze the structure of the product to determine characteristics about it. Software complexity and data flow research fit under this model” [136].

Static analysis is used in this research by analyzing the architectures of numerous legacy systems. This was done to increase understanding of the use of architecture patterns. It was used in the following research question:

- **RQ-1:** *How common are architecture patterns in software architectures? In particular, which patterns are commonly found individually and in pairs in certain application domains?* This research analyzed architecture documentation of numerous systems, and is given in chapter 2.

• **Exploratory Survey:** this method is a Field Study, which “involves experimental design but no experimental controls, [and] is carried out in the natural settings of the phenomenon of interest” [3]. An Exploratory Survey is such a study “in which there is no test of relationships between variables” [57].
In this research, an exploratory survey was done to understand the effectiveness of a method of using architecture patterns in software architecture reviews. This is contained in the following research question:

- **RQ-6: How can one gain insight into the impact of the architecture patterns used on quality attributes early in the development cycle – while the architecture can still be readily modified?** In addition to developing an architecture review method, this research employed an experimental study to evaluate its cost and effectiveness (see chapter 8).

- **Laboratory Experiment:** this method is an experiment in a controlled setting, involving manipulating an independent variable, with controls for intervening variables [3]. This was used in gathering information for the following research question:

- **RQ-3a: What do we learn about patterns and satisfying quality attributes through the application of this model?** We created a controlled experiment in which we compared architectural activities between a group with pattern-tactic interaction information and a control group without it.

### 1.6.3 Types of Research Questions:

Shaw describes several types of research questions that may be addressed in software engineering research. In this thesis, two types of research questions are posed:

- **Generalization or Characterization:** These questions aim to increase the understanding of the topic. They include questions about the meaning and characteristics of the topic (e.g., “what exactly do we mean by X?”, “What are the important characteristics of X?”) They can include questions about taxonomies associated with the topic (e.g., what are the varieties of X, and how are they related?”) They may also include questions about models that represent the topic (e.g., “What is a good formal or empirical model of X?”) Much of this work is devoted to increasing the breadth and depth of insight into software architecture and patterns; thus many of the research questions are of this type.

- **RQ-1** aims to characterize the typical use of architecture patterns

- **RQ-2** aims to characterize how architecture patterns and decisions are related, particularly those concerned with quality attributes
- RQ-3 aims to create a general model of the interaction of patterns and tactics.
- RQ-3a aims to characterize the nature and impact of the application of the model from RQ-3.
- RQ-4 aims to create a general model of this interaction when multiple patterns and tactics are involved.

- **Method or Means of Development**: These questions explore ways to improve the practice of software development. They include questions about performing or automating software development tasks, and questions about improving existing ways of doing them.
  - RQ-5 searches for a method for architectural analysis and synthesis.
  - RQ-6 searches for a method for architectural evaluation.

### 1.6.4 Research Result Types

What types of results do the research questions have? Shaw identifies several types of results of research to answer the research questions. The types of results found in this research are as follows:

- **Answer or Judgment**: This type is the result of a specific analysis, evaluation, or comparison.
  - The results of RQ-1 are answers about pattern usage based on analysis of architectures.
  - The result of RQ-2 is a judgment based on the comparison of patterns, to architecture decisions, and how they fit with quality attributes.
  - The result of RQ-3a is a judgment of what can be learned through use of the model of patterns and tactic.

- **Qualitative or Descriptive Model**: This result is a structure or taxonomy for a problem area, such as architectural styles, frameworks, etc. It may also include well-grounded checklists, well-argued informal generalizations, or guidance for integrating other results.
  - The result of RQ-3 is a model that describes the relationship of patterns and tactics.
  - The result of RQ-4 is an extension of that model.
• **Procedure or Technique**: These results show a new or better way to do some task, such as design, implementation, measurement, evaluation, selection from alternatives, etc. It may include operational techniques for implementation representation, or analysis. It does not include advice or guidelines (see qualitative or descriptive model, above.) The results of research questions RQ-5 and RQ-6 are of this type.

  - The result of RQ-5 is an architectural analysis and synthesis technique.
  - The result of RQ-6 is an architectural evaluation technique.

### 1.6.5 Validation Techniques

Research results require convincing evidence that the results are sound. There are various types of validation used in software engineering research to demonstrate the validity of the research results. Of the various types of validation described by Shaw, this research employs the following methods:

- **Evaluation, Descriptive Model**: The result adequately describes the phenomena of interest. The research results concerning RQ-1 are validated through the study of 47 software architectures from industry. Research question RQ-4 is validated by describing the use of architecture patterns and analyzing their use and interaction with tactics, as well as with each other.

- **Evaluation, Qualitative Model**: The result accounts for the phenomena of interest. Research question RQ-2 is validated by describing the use of patterns and comparing them to architectural decisions as well as their interaction with quality attributes.

- **Example, Slice of Life**: Slice of Life examples show an example of how the research is valid using a real setting. Research question RQ-3’s results are validated by applying the proposed model against several architectures taken from industry.

- **Analysis, Empirical Model**: Analysis of the data is accomplished through empirical validation. Results concerning research questions RQ-3a and RQ-6 are validated through empirical validation. Validation of RQ-3’s results is accomplished through a controlled experiment of subjects performing architectural design. Results concerning research question RQ-6 is validated through analysis of data in an exploratory study [57] using nine real projects.

- **Example, Toy Example**: An example shows an example of how the research is valid using a contrived example designed to mimic the real world. Results for research question RQ-5 gives an example of use of the architecture method
The example describes a hypothetical system under design, and walks through each step of the architecture method, showing the use of architecture patterns in the steps.

Table 1.1 summarizes the research question types, the research results, and validation types used in this thesis.

Table 1.1: Research types Summary

<table>
<thead>
<tr>
<th>RQ</th>
<th>Research Method Used</th>
<th>Research Question Type</th>
<th>Research Result</th>
<th>Validation Technique</th>
<th>Thesis Chapters</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ-1</td>
<td>Static Analysis</td>
<td>Generalization or characterization</td>
<td>Answer or judgment</td>
<td>Evaluation, descriptive model</td>
<td>2</td>
</tr>
<tr>
<td>RQ-2</td>
<td>Descriptive Research</td>
<td>Generalization or characterization</td>
<td>Answer or judgment</td>
<td>Evaluation, qualitative model</td>
<td>3</td>
</tr>
<tr>
<td>RQ-3</td>
<td>Descriptive Research</td>
<td>Generalization or characterization</td>
<td>Qualitative or descriptive model</td>
<td>Example, slice of life</td>
<td>4</td>
</tr>
<tr>
<td>RQ-3a</td>
<td>Laboratory Experiment</td>
<td>Generalization or characterization</td>
<td>Answer or judgment</td>
<td>Analysis, Empirical model</td>
<td>5</td>
</tr>
<tr>
<td>RQ-4</td>
<td>Descriptive Research</td>
<td>Generalization or characterization</td>
<td>Qualitative or descriptive model</td>
<td>Analysis, Empirical model</td>
<td>6</td>
</tr>
<tr>
<td>RQ-5</td>
<td>Developmental Research</td>
<td>Method or means of development</td>
<td>Procedure or technique</td>
<td>Example, toy example</td>
<td>7</td>
</tr>
<tr>
<td>RQ-6</td>
<td>Developmental Research</td>
<td>Method or means of development</td>
<td>Procedure or technique</td>
<td>Analysis, empirical model</td>
<td>8</td>
</tr>
</tbody>
</table>
1.7 Thesis Overview

The main body of this thesis is based on publications in journals (chapters 3, 4, and 8), a book chapter (chapter 6), and international conferences (chapters 2, 3, 6, and 8). The material in the chapters is taken from the publications with only trivial changes, except that introductory material has been removed where it is already covered. A summary of each chapter follows:

**Analysis of Architecture Pattern Usage in Legacy System Architecture**

**Documentation** [62]: Architecture patterns are an important tool in software architecture design. However, while many architecture patterns have been identified, there is little in-depth understanding of their actual use in software architectures. In addition, little is known of how architecture patterns interact with each other. In order to learn more about these topics, we studied the architecture documentation of 47 systems. We found that most systems used two or more architecture patterns, and that certain patterns were prominent in different application domains.

This chapter makes the case for utility of this research – architecture patterns are indeed pervasive in industrial software systems, therefore the study of architecture patterns can potentially have widespread benefit. In addition, the fact that different patterns were used in different application domains suggests that the patterns may be tied to the characteristics of the domain, such as the quality attributes that are important in a domain.

**Chapter: 2, Research Questions: RQ-1**

**Leveraging Patterns to Satisfy Quality Attributes and Document Architectural Decisions** [64, 70]: All of software design involves making decisions and reifying those decisions in code. The decisions made during software architecting are particularly significant, because they have system-wide implications. In particular, they can impact the ability of the software to meet its quality attribute requirements. However, the impact may be first understood much later, when the architecture is difficult to change. In addition, architectural decisions are often not well documented, which leads to architectural erosion.

We found that architecture patterns are a common solution to both these problems. They can help architects understand the impact of architectural decisions at the time the decisions are made, because many patterns contain information about the consequences of using the pattern. At the same time, they are an important way to capture and recover knowledge of key architectural decisions. This research helps broaden the understanding of the
utility of architecture patterns. It also helps substantiate the relationship between the patterns and quality attributes.

Chapter: 3, Research Questions: RQ-2

How do Architecture Patterns and Tactics Interact? A Model and Annotation [65]: In the previous chapters we showed that patterns interact with tactics used to implement quality attributes. In this chapter, we explore the nature of this interaction. We found that quality attributes are satisfied through the implementation of specific measures which are called tactics. Therefore, the impact of patterns on quality attributes is through the interaction of the patterns with the tactics. We defined a model of this interaction. We defined types of changes to patterns’ structure and behaviour that the implementation of a tactic may cause. These changes can have minor or major impact on the pattern, which indicates how readily the tactic can be implemented in the pattern. This information gets to the root of the question of why an architecture pattern may be more or less compatible with a given quality attribute.

Chapter: 4, Research Questions: RQ-3

Applying a Pattern-Tactic Interaction Model in one Quality Attribute Domain [65, 71]: In this chapter, we apply the model of interaction of patterns and tactics to tactics commonly used to achieve fault-tolerant systems. This study showed that certain patterns are very supportive of fault tolerance tactics, while other patterns are generally incompatible with them. We explored the benefits of applying the model by performing an experiment in software architectural design. Two teams were asked to add designs for certain fault tolerance tactics. One team was given architectural pattern information, along with information about the interaction of the patterns with common fault tolerance tactics; the other was not. The team with the interaction information was better able to design the system for the tactics, and better able to understand the impact on the architecture. This supports the notion that the pattern-tactic interaction information is useful for architects.

Chapter: 5, Research Questions: RQ-3a

The Interaction of Requirements, Tactics and Architecture Patterns [66]: Here we research the interaction of architecture patterns and tactics where there are multiple architecture patterns and/or multiple tactics in play. We found that some tactics require implementation in all components of the system, while others can be implemented just in certain parts of the system. The location of the implementation of these tactics is influenced by first requirements of the
domain, and second the compatibility of the patterns with the tactic. This gives more detail about how they interact in complex systems.

Chapter: 6, Research Questions: RQ-4

**Pattern-Driven Architectural Partitioning: Balancing functional and Non-functional Requirements** [63]: This research lays out an architectural design process that incorporates architecture patterns into the process. It leverages patterns to help the architect make decisions about how to fulfill the quality attribute (non-functional) requirements of the system. The process is compatible with a well-known general architectural design process. The paper gives an example of how it can be used.

Chapter: 7, Research Questions: RQ-5

**Using Pattern-Based Architecture Reviews to Detect Quality Attribute Issues** [67, 68]: In this chapter we explored the use of architecture reviews based on the patterns used to find potential issues with satisfying quality attributes. This study indicates that such reviews can help identify potential problems well before the code is written. These problems are identified in time to make necessary architectural changes before the architecture is set in stone. Furthermore, the study showed that for small projects, the reviews require only small amounts of time and effort, making them feasible for projects that until now have not been able to take advantage of architecture reviews because of their high cost.

Chapter: 8, Research Questions: RQ-6

The final chapter of the thesis summarizes the research by presenting the research questions and answers, and describing the key contributions of this work. As no scientific research is ever completely finished, this thesis concludes with open questions and suggestions for further investigation. I hope that these questions will stimulate additional research that furthers the progress of software architecture.