Spring 6-13-2014

Preserving the Quality of Architectural Tactics in Source Code

Mehdi Mirakhorli
mirakhorli@gmail.com

DEPAUL UNIVERSITY

Preserving the Quality of Architectural Tactics in Source Code

by

Mehdi Mirakhorli

Dissertation Thesis submitted in partial fulfillment for
the degree of Doctor of Philosophy

in the
School of Computing
College of Computing and Digital Media

Advisor:
Jane Cleland-Huang, PhD

June 2014


architectural
traceability
tactic
tactics
decisions
design
links
Hadoop
Figure
software
system
data
classes
Approach
code
Tactic
Methodology
tTIMs
heartbeat
using
used
Traceability
architecture
Chapter
results
tTIM
roles
classifier
systems
Heartbeat
quality
approach
Software
specific
set
research
requirements
study
different
HDFS
maintenance
classification
experiment
case
related
across
also
level
two
decision
therefore
use
support
tactic-related
Studies
threshold
emitter
following
models
provides
Requirements
Information
identified
Threshold
implementation
trace
View
Name
meta-model
example
goal
Data
Table
result
work
process
first
proposed
Robot
designed
change
security
tracing
one
provide
scheduling
patterns
Appendix
snippets
task
Decisions
projects
CEV
implemented
Case
concerns
terms
various
must
visible
performance
information
number
class
created
tactical
Lunar
Proceedings
Tracker
goals
analysis
identify
developers
Classification
Architecture
component
underlying
components
cluster
notifications
Design
failure
role
term
shows
cases
message
often
pp
Node
order
training
GN
user
experiments
Architectural
three
five
Code
tasks
blocks
Scheduling
method
F-Measure
Vehicle
conducted
block
node
model
Model

DEPAUL UNIVERSITY
School of Computing
College of Computing and Digital Media

Abstract

Preserving the Quality of Architectural Tactics in Source Code

by Mehdi Mirakhorli

In any complex software system, strong interdependencies exist between requirements and software architecture. Requirements drive architectural choices while also being constrained by the existing architecture and by what is economically feasible. This makes it advisable to concurrently specify the requirements, to devise and compare alternative architectural design solutions, and ultimately to make a series of design decisions in order to satisfy each of the quality concerns.

Unfortunately, anecdotal evidence has shown that architectural knowledge tends to be tacit in nature, stored in the heads of people, and lost over time. Therefore, developers often lack comprehensive knowledge of underlying architectural design decisions and inadvertently degrade the quality of the architecture while performing maintenance activities. In practice, this problem can be addressed through preserving the relationships between the requirements, architectural design decisions and their implementations in the source code, and then using this information to keep developers aware of critical architectural aspects of the code.

This dissertation presents a novel approach that utilizes machine learning techniques to recover and preserve the relationships between architecturally significant requirements, architectural decisions and their realizations in the implemented code.

Our approach for recovering architectural decisions includes the two primary stages of training and classification. In the first stage, the classifier is trained using code snippets of different architectural decisions collected from various software systems. During this phase, the classifier learns the terms that developers typically use to implement each architectural decision. These "indicator terms" represent method names, variable names, comments, or the development APIs that developers inevitably use to implement various architectural decisions. A probabilistic weight is then computed for each potential indicator term with respect to each type of architectural decision. The weight estimates how strongly an indicator term represents a specific architectural tactics/decisions. For

example, a term such as *pulse* is highly representative of the heartbeat tactic but occurs infrequently in the authentication. After learning the indicator terms, the classifier can compute the likelihood that any given source file implements a specific architectural decision.

The classifier was evaluated through several different experiments including classical cross-validation over code snippets of 50 open source projects and on the entire source code of a large scale software system. Results showed that classifier can reliably recognize a wide range of architectural decisions.

The technique introduced in this dissertation is used to develop the Archie tool suite. Archie is a plug-in for Eclipse and is designed to detect wide range of architectural design decisions in the code and to protect them from potential degradation during maintenance activities. It has several features for performing change impact analysis of architectural concerns at both the code and design level and proactively keep developers informed of underlying architectural decisions during maintenance activities.

Archie is at the stage of technology transfer at the US Department of Homeland Security where it is purely used to detect and monitor security choices. Furthermore, this outcome is integrated into the Department of Homeland Security's Software Assurance Market Place (SWAMP) to advance research and development of secure software systems.

"This Thesis is dedicated to the memory of my parents"

Acknowledgements

My advisor, Professor Jane Cleland-Huang, has had the most profound influence on me as a researcher. She gave me the freedom to explore on my own while never ceasing to challenge me. I have found working with Professor Jane Cleland-Huang very rewarding, and I believe I have learned the research styles, manners and ethics from the best. Her inspiration and confidence in me have afforded me many opportunities to interact with the most accomplished researchers in our field on an equal footing. Jane is truly more than an advisor, and I could not have asked for a better role model. I am sure I will miss walk into her office to talk and receive priceless advice.

I also gratefully acknowledge the constructive feedback and excellent input that I received from Dr. Bamshad Mobasher and Dr. Xiaoping Jia on an earlier version of this thesis. Their advice helped me to improve the quality of my work.

During last two years, I established a great working relationship with Dr. Mark Grechanik. He has always been ready to engage in long research discussions and to provide his unique insight. I hope this relationship continues in future.

I would also like to show my gratitude to my friends and collaborators, Dr. Roshanak Roshandel, Dr. Patrick Maeder and Dr. Sam Malek, who helped me countless times to sort through things both professional and personal.

I owe my deepest gratitude to my parents, for teaching me the value of education and encouraging me to follow my dreams.

The work in this thesis has been partially funded by the National Science Foundation (NSF) (Num: 1218303) and the US Department of Homeland Security (DHS).

Contents

List of Figures

[1] footnote
[2] footnote2

List of Tables

Abbreviations

ASR	**A**rchitecturally **S**ignificant **R**equirements
ATAM	**A**rchitecture **T**rade-off **A**nalysis **M**ethod
CMMI	**C**apability **M**aturity **M**odel **I**ntegration
FAA	**F**ederal **A**viation **A**dministration
FDA	**F**ood and **D**rug **A**dministration
DOD	**D**epartment **O**f **D**efence
DCT	**D**ecision **C**entric **T**raceability
EBT	**E**vent **B**ased **T**raceability
TIM	**T**raceability **I**nformation **M**odel
tTP	**t**actic **T**raceability **P**attern
NFR	**N**on-**F**unctional **R**equirements
IDE	**I**ntegrated **D**evelopment **E**nvironment
IRB	**I**nternal **R**eview **B**oard
SAREC	**S**oftware **a**nd **R**equirements **E**ngineering **C**enter
COEST	**C**enter **o**f **E**xcellent for **S**oftware **T**raceability

Part I

Problem Statement and Background

"It takes a lot of courage to show your dreams to someone else."

Erma Bombeck

Chapter 1

Introduction

Nowadays, a successful software production is increasingly dependent on how the final deployed system addresses customers' and users' quality concerns such as security, reliability, availability, interoperability and performance. Failure to address these qualities in a best case scenario, could result in a significant drop in the number of users, negatively impact the market value of an enterprise which uses the software, consequently result in the loss of value in stock price. In more extreme cases it could even lead to catastrophic and systematic failures threatening public safety. The media is full of reports of the impact of software failure. For example a software failure interrupted the New York Mercantile Exchange and telephone service to several East Coast cities in February 1998. In another recent example, one Illinois hospital jointly managed by the Departments of Veterans Affairs ("VA") and Defense ("DOD") failed to achieve interoperability between the Departments' EHR systems, costing the hospital at least $700,000 annually. This is despite the fact that the DOD and VA had already spent $100 million to achieve this quality.

In order to satisfy such quality concerns, software architects are accountable for devising and comparing various alternate solutions, assessing the trade-offs, and finally adopting strategic design decisions which optimize the degree to which each of the quality concerns is satisfied. The adopted decisions are often based on well known architectural *tactics*, defined as re-usable techniques for achieving specific quality concerns [81]. Bachman et al. provide a more formal definition for an architectural tactic as "a means of satisfying a quality-attribute-response measure by manipulating some aspects of a quality attribute model through architectural design decisions" [7]. Architectural

tactics come in many different shapes and forms and describe solutions for a wide range of quality concerns [64, 81]. For instance, reliability tactics provide solutions for fault detection, prevention and recovery; performance tactics provide solutions for resource contention in order to optimize response time and throughput, and security tactics provide solutions for authorization, authentication, non-repudiation and other such factors [64]. Chapter 2 provides a comprehensive overview of some common architectural tactics.

Traditionally, software architecture has been seen as the "structure or structures of the system". From this perspective an architecture is defined as a set of software components, their externally visible properties, and their interconnections [81]. However this view, addresses only the physical infrastructure of a system, and fails to capture the importance of other architectural decisions. As a result, the more modern definition by Bosch [19, 72], Kruchten [78], Perry [100] and others emphasizes software architecture as a set of interrelated design decisions. These decisions can be technology related, structural for shaping the skeleton of the system, process related, or they could be related to governance issues. From this perspective, architectural quality is achieved not only through traditional engineering practices such as module decomposition, information hiding and abstraction [48, 119], but also through managing and preserving a broad set of tactical architectural decisions.

Despite the tremendous effort that goes into designing and delivering a robust architectural solution, architectural qualities are often eroded over time as a result of ongoing maintenance activities, which are inevitably undertaken to correct faults, improve performance or other quality concerns, and to adapt the system in response to changing requirements [71, 127]. In such a scenario, when the software product evolves in subsequent releases, maintainability of the system decreases, making it difficult to understand and modify. In the best case, changes accumulate and become obstacles for further releases, while in a severe case the architecture degradation can lead to catastrophic failure. This phenomenon has been referred to as *Architecture Erosion* [87][86][127][100][125]. Netscape browser is a good example of architecture erosion [127]. In early 1997, Netscape and Microsoft were in a browser war, where Netscape released Netscape Communicator version 4.0 and at the same time Microsoft sent out its version 4.0 of Internet Explorer. A year later, for various reasons Netscape was losing the competition. Therefore they urgently decided to move toward an improved version of their Netscape Communicator to beat Microsoft. However, they could not deliver it. The

architecture of the old version 4 was eroded and the source code was hard to modify. Finally in late 1998, Netscape, in a desperate attempt to regain its lost market share, decided to release its browser as open source. After a few months, the old source code was discarded entirely and development of a new browser (known as Mozilla) was started.

This is a challenging problem of software development recognized by both academics and practitioners. Perry and Wolf first drew attention to the problem in 1992 [100]. They defined architectural *erosion* as 'violations of the architecture' and architectural *drift* as an 'insensitivity to the architecture' which occurs when the underlying rules are not clear to the developers and maintainers. The problem is far from a theoretical one and unfortunately has not been addressed properly. Most popular software engineering tools and environments fail to make underlying design decisions visible to software engineers, and as a result maintainers are not kept fully informed of relevant underlying design patterns, tactics, and constraints as they construct, maintain, and refactor a software system [18][127].

In practice, the architecture of a system and its qualities can be maintained using traceability methods that help developers fully understand the impact of design or implementation changes on architecturally significant requirements [127].

In related work, Kruchten [78], Burge [23], and others have proposed a proactive approach to preventing design degradation through using design rationales to document architectural decisions. They argue that explicitly recording design decisions, justifications, alternatives, and conflicting perspectives, is necessary in order to preserve architectural qualities. Several researchers have attempted to address this problem through developing techniques for documenting or modeling architectural decisions. The Architecture Design Decision Support System (ADDSS) [25], Process based Architecture Knowledge Management Environment (PAKME)[13], and Architecture Rationale and Element Linkage (AREL)[118] are examples of these. However, these approaches fail to address the scalability issues of managing potentially large numbers of architectural decisions. They also fail to connect design decisions to code, and/or provide little support for actually utilizing this knowledge during software maintenance.

Moreover, architecture erosion is very likely to occur if developers are 'insensitive to the architecture' [100] and the architecturally implied decisions/rules are not clear to them, or when the developers lack experience and therefore misunderstand the key architectural decisions or do not

have enough knowledge to implement/modify architectural decisions in a robust and optimum way [127]. Although there have been numerous books, materials and tutorials on architectural tactics, most of these are at a high level of design and provide insufficient guidance for how to implement these tactics.

This dissertation, primarily suggests using Software Traceability, defined as the "the ability to interrelate any uniquely identifiable software engineering artifact to any other, maintain required links over time, and use the resulting network to answer questions of both the software product and its development process" [33] and to establish links between uniquely identifiable and related software engineering artifacts, as a means of preventing erosion. It proposes the new notion of decision-centric-traceability which can potentially address the aforementioned problems through explicitly documenting relationships between quality concerns, rationales, architectural decisions, and source code. This technique puts architectural decisions (such as tactics) as the focus of the tracing activity. In fact, in this approach the driving requirements, design rationale and constraints are first traced to their corresponding architectural tactics, and tactics are then used to link the requirements knowledge to the implementation artifacts. Such links can then be used to support change impact analysis and program comprehension during the maintenance process by revealing underlying design decisions. In order to trace each architectural tactic in an economical way, this dissertation introduces the concept of tactic traceability information models, designed to guide the creation of sufficient traceability links.

Secondly, this dissertation, proposes automated reconstruction of traceability links between quality concerns and source code through using machine learning techniques to detect architectural tactics in the code. Considering the acceptable level of accuracy achieved by this technique, it would significantly minimize the cost of establishing and maintaining traceability links by dynamically creating the links upon demand by developers.

Finally the third part of the dissertation, provides guidelines for developers, presumably less experienced ones, on how to implement architectural tactics by utilizing robust and effective design patterns which tend to be more maintainable. These guidelines are extracted through observations of various tactic implementations. First the tactical code was detected in a large number of software systems, then various design patterns were detected in the software systems. Overlap analysis was performed to find evidence of design patterns being used to implement a tactic. The results of this

analysis are provided though a reference model for each tactic. This reference model allows the developers to select one or more appropriate design patterns to implement a tactic based on the context of their projects.

1.1 Contributions

This dissertation makes the following primary contributions:

• Identify Traceability Challenges of Architectural Decisions

Before proposing any specific guidelines for tracing quality concerns, it is necessary to understand the issues that must drive any effective architecture traceability solution. The first part of this dissertation presents the results of an extensive study conducted on tactical architectural decisions implemented in highly dependable and complex avionic systems such as AirBus and Boeing family, NASA Robotics, NASA Crew Exploration Vehicle (CEV), Hadoop Framework and various other performance centric systems. This study first identifies the key challenges of tracing architecturally significant requirements and then provides the foundations and motivation for a new traceability meta-model describing a traceability infrastructure for efficiently tracing individual quality concerns.

• Identifying Traceability Patterns for Tactics

The previously designed meta-model is extended in the form of the potentially transformative notion of a Tactic Traceability Pattern (tTP), defined as an instantiable and reusable tactic-centric traceability model. A tTP can reduce traceability effort by providing a reusable infrastructure for tracing, detecting, managing, and preserving tactics throughout the maintenance phase of the software system. Chapter 4 provides a more detailed description of the tTP concept and describes a set of tTPs developed for this dissertation.

• Automating Trace Generation

Manually creating trace links is a difficult and expensive task. Therefore we have developed a novel approach for automating the generation of trace links between tactics and the design and code elements in which they are implemented. Chapter 5 describes this approach and presents the results of its evaluation. The developed approach uses and trains a classifier to detect architectural tactics in

the source code and to subsequently establish trace links to the tTPs. Trace links are established at two levels of granularity, Tactic-Level and Role-Level. The classifier is trained through two different training methods which either use a data set of *tactical code-snippets* or *tactic-descriptions* to build the classifier. Furthermore, in order to create trace links at the Role-Level we have developed a hybrid approach which utilizes a light-weight structural analysis technique to create fine-grained trace links to the tactics' roles in a tTP. Although the results are not as good as those obtained for Tactic-Level link reconstruction, the output can still be used to help developers create links.

The major contribution of this work to the existing body of knowledge is the focus on architectural tactics. This can be viewed from two perspectives. All the bulk of existing research on software architecture discovery, recovery and reconstruction from source code has primarily focused on modularization and structure of software. The novel part of our work is taking into account tactical architectural decisions which are pervasive types of decisions in various systems. Also from the perspective of software traceability this is the first work that has focused on automating the traceability of architectural tactics into the source code.

•Informed Notification and Visualization of Design Rationale

In an IEEE Software article entitled "Draw me a picture", Grady Booch [18] challenged the software engineering community to develop new visualization tools capable of providing greater insights into underlying frictions, design decisions, and social factors of a software system. Our traceability approach makes an initial and partial contribution to addressing this challenge through notifying the developer of underling decision and showing how different parts of the system work together to achieve various quality goals. This can help reduce the risk of design erosion by keeping developers informed of tactical architectural decisions behind the code and notifying them which architectural tactics and related software qualities can be affected by the changes they implement. This would address one of the key causes of architectural erosion known as insensitivity to design. The notification mechanism is built primarily on top of our automated trace reconstruction technique.

• Presenting Reference Models for Implementing Tactics

Once a decision is made to utilize a tactic, the developer must generate a concrete plan for realizing the tactic in the code. Unfortunately, there is not a single way to implement an architectural tactic. From one system to another system a tactic can be implemented entirely differently and this is

according the context and constraints of each project. The variability points found in individual tactics can make this a challenging task, especially for less experienced developers. This is a typical knowledge gap that exists between high level architecture design and low level programming as the scope of concerns are different. To address this knowledge gap, we conducted an extensive study of 50 open source systems to investigate how design patterns were used to implement various tactics. Data mining techniques were used to identify potential pattern instances within tactic implementations. This was followed by a manual analysis of the retrieved data to identify a distinct set of variability points for each tactic, as well as evidence of corresponding design patterns used to address them.

The output of this observation resulted in the construction of tactic-level decision trees depicting variability points of a tactic and generating a reference model which provides implementation guidance for the developers.

This perspective, recognizes Parnas' notion of "design for change" [98][97]. In this perspective, architectural erosion is tackled by providing guidance to help developers implement or modify tactical decisions through the use of known design patterns. The use of design patterns is considered more robust and maintainable way for low level implementation and tend to be more flexible for evolution while at the same time are harder to erode [67][129][52].

1.2 Overview of the Methodology

In order to achieve each goal identified in the previous section, we apply a research methodology which has three phase of *problem analysis*, *solution design* and *solution validation*.

In problem analysis we explore various challenges for addressing each of the research goals. The problem analysis includes extensive studies of key challenges in the real systems from various domains of Avionics, Business and Financial platforms, Distributed Computing Frameworks, Internet Based Applications and etc. This has provided a realistic foundation for understanding the problem. In the next phase, the gained insight in the problem analysis is used to drive an appropriate solution. In the validation phase we examine the proposed solution and rigorously validate it through several experiments.

1.3 Scope

As the scope of architectural qualities and their implementation is vast, we limit the scope of the thesis to a subset of architectural decisions and primarily those known as tactics. The goal of this dissertation is to demonstrate that solutions for preventing architectural erosion exist however it is not intended as a survey of all known architectural design decisions. That is left for future research.

As part of this work, a set of information retrieval and data mining techniques have been applied, however the purpose of the experiments in this dissertation is not to find the best available algorithm, compare existing ones, or improve known methods. Instead, this dissertations investigates areas in software architecture development in which such methods can be used as tools to help provide useful support to cope with the problem of erosion.

Although we have proposed a catalogue of tTPs, the contribution of this dissertation, is not in providing a fully validated set of tTPs, but rather in showing how a tTP can be used to support effective software maintenance while reducing the likelihood of architectural degradation.

While the scope of the thesis was limited to what is mentioned above, any results that were potentially applicable to a wider scope are indicated throughout the work.

1.4 Published Work

This dissertation includes work published in the following international peer-review workshops, conferences and journals:

- Kouroshfar, E.; Mirakhorli, M.; Bagheri, H.; Xiao, L.; Malek, S.; Cai, Y.; "A Study on the Role of Software Architecture in the Evolution and Quality of Software", *Submitted to the ACM SIGSOFT 22th International Symposium on the Foundations of Software Engineering*, 2014,ACM.
- Cleland Huang, M.; Ali Babar, M.; Mirakhorli, M.; "An Inverted Classroom Experience: Engaging Students in Architectural Thinking for Agile Projects", *Software Engineering Education and Training (SEET) Track, IEEE International Conference on Software Engineering (ICSE'14)*.

- Cleland Huang, M.; Czauderna, A.; Mirakhorli, M.; "Driving Architectural Design and Preservation from a Persona Perspective in Agile Projects", *Agile Software Architecture*, edited by Muhammad Ali Babar, Ivan Mistrik, and Alan Brown, 2014.

- Mirakhorli, M.; "Preventing Erosion of Architectural Tactics through Their Strategic Implementation, Preservation, and Visualization", *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013.

- Mirakhorli, M.; Carvalho, J.; Cleland-Huang, J.; Maeder, P.; "A Domain-centric Approach for Recommending Architectural Tactics to Satisfy Quality Concerns", *Third International Workshop on the Twin Peaks of Requirements and Architecture*, 21st IEEE International Requirements Engineering Conference (RE'13), 2013.

- Mirakhorli, M.; Cleland-Huang, J.; "Traversing the Twin Peaks", *IEEE Software*,vol.30, no.2, March-April 2013.

- Mirakhorli, M.; Maeder, P.; Cleland-Huang, J.; "Variability points and design pattern usage in architectural tactics", *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 52,2012,ACM.

- Mirakhorli, M.; Shin, Y.; Cleland-Huang, J.; Cinar, M.; "A tactic-centric approach for automating traceability of quality concerns", *Proceedings of the 2012 International Conference on Software Engineering*, 639-649,2012,IEEE Press. **ACM SIGSOFT Distinguished Paper Award**

- Cleland-Huang, J.; Mirakhorli, M.; Czauderna, A.; Wieloch, M.; "Decision-Centric Traceability of Architectural Concerns", *The 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2013)*.

- Mirakhorli, M.; Cleland-Huang, J.; "Using tactic traceability information models to reduce the risk of architectural degradation during system maintenance", *Software Maintenance (ICSM), 2011 27th IEEE International Conference on Software Maintenance*, 123-132,2011,IEEE.

- Mirakhorli, M.; Cleland-Huang, J.; "A pattern system for tracing architectural concerns", *Proc. of the Pattern Languages of Programming Languages (PLOP)*, 2011.

- Mirakhorli, M.; Cleland-Huang, J.; "Tracing Non-Functional Requirements", *Software and Systems Traceability*, 299-320,2012,Springer

- Mirakhorli, M.; "Tracing architecturally significant requirements: a decision-centric approach", *33rd International Conference on Software Engineering*, 1126-1127,2011,IEEE.

- Mirakhorli, M.; Cleland-Huang, J.; "Tracing architectural concerns in high assurance systems (NIER track)",*Proceedings of the 33rd International Conference on Software Engineering*,908-911,2011,ACM.

- Mirakhorli, M.; Cleland-Huang, J.; "A decision-centric approach for tracing reliability concerns in embedded software systems",*Proceedings of the Workshop on Embedded Software Reliability (ESR)*, held at ISSRE10, 2010.

- Mirakhorli, M.; Cleland-Huang, J.; "Transforming trace information in architectural documents into re-usable and effective traceability links", *Proceedings of the 6th International Workshop on SHAring and Reusing Architectural Knowledge*, 45-52,2011,ACM.

1.5 Organization

This dissertation is organized as follows: The Part I includes the literature review presented in Chapters 2 and 3. Part II of this dissertation focuses on creating architecture traceability. In Chapter 4 we present the traceability challenges of architectural decisions that we have identified as a result of studying various real systems; Also illustrates the idea and utilization of Traceability Patterns for Tactics. Chapter 5 presents the approach and results for automating the generation of trace links from architectural tactics to source code. Chapter 6 presents a ranking comparison of our tactic detector approach with a number of Off-The-Shelf text categorization methods as well as a voting approach including all the classification methods. In Part III we will show how to use the previously developed approach to help reduce the risk of design erosion by keeping the developers informed of tactical architectural decisions behind the code.

Part IV presents our tactic reference models developed to support tactic's implementation and modification.

Finally, Part V and chapter 9 presents the conclusions of the thesis, the main threats to validity, as well as future work that can be done to augment it. All the case studies used through this dissertation are presented in Appendix A.

"All the world's a stage,
And all the men and women merely players:
They have their exits and their entrances;
And one man in his time plays many parts;"

William Shakespeare

Chapter 2

Background and Definitions

In our work, we utilize concepts and techniques from various areas of software engineering and computer science such as software architecture, requirements engineering, software traceability, and data mining and information retrieval. In this chapter, we provide background information on these areas and introduce a set of definitions used throughout the thesis.

The first section of this chapter presents various definitions for the concept of *Software Architecture* followed by a description of Architectural Tactics as common approaches to design an architecture in section 2.2. In Section 2.3, a set of real systems implementing architectural tactics are discussed. Section 2.4 describes the phenomena of architecture erosion which makes software qualities degrade over time. This section presents the common causes of such phenomena and describes the state of art and practice for dealing with such problems.

2.1 Software Architecture

Every software development process or life-cycle is structured around a set of classical disciplines represented as *Requirements Engineering*, *Software Design*, *Implementation*,*Testing* and *Maintenance*. Following the principles and practices in each of these disciplines results in the creation of a robust working product from an initial customer request. While the requirements engineering discipline helps analysts to elicit, analyse and understand the customers concerns and users needs, the design discipline leads to the development of abstract solutions to transit from such requirements

to implementations and a working product. In this process, Software Architecture Design occurs at the overlap of the two aforementioned disciplines. It includes, principles and practices of both requirements engineering and software design. Over the last decade, software architecture design has increasingly been considered as a key activity to influence software qualities such as security, reliability, performance and so on. Therefore it has been the focus of a wide range of research and practice.

2.1.1 Definitions of Software Architecture

There are various definitions for architecture of a software system. Traditional definitions consider software architecture as the structure or skeleton of a system. In this definition, architecture is a collection of components building a software system. For example, Len Bass [81] defines software architecture as

> *"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".*

Similarly Mary Shaw and David Garlan [55] describe software architecture as:

> *"...a level of design concerned with issues beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives".*

The IEEE 1471-2000 standard [60] has defined the term *Architecture* as:

> *"The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution"*

Additionally, this standard provides a description for each of the terminologies used in this definition of architecture. A *system* is a collection of components organized to implement a specific set of functions. The term system is very broad and can include individual applications, subsystems, systems of systems, products and product families, whole enterprises, and other such interests.

The *environment*, is in fact the context and determines the setting for developmental, operational, political, and other factors influencing the system [60].

A *stakeholder* in this definition is an individual, team, or organization with interests in a system [60].

A key criticism to all the above definitions is that they address only the physical infrastructure of a system, and fail to capture the importance of other architectural decisions. The proponents of alternative definitions Bosch [8], Kruchten [9], Perry [10] and others refer to software architecture as a set of interrelated design decisions which work together to shape the structure, behavior, properties, processes, and governance of the delivered solution. Among these, Bosch argues that,

> *"we define software architecture as the composition of a set of architectural design decisions. This reduces the knowledge vaporization of design decision information, since design decisions have become an explicit part of the architecture".*

Kruchten in The Rational Unified Process 1999, defines architecture as:

> *"An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization—these elements and their interfaces, their collaborations, and their composition".*

From this perspective, architectural quality is achieved not only through traditional engineering practices such as partitioning and abstraction [119][48], but also through managing and preserving a broad set of architectural decisions.

Analysis of different architectural decisions indicate that these decisions appear in different shapes and forms. Sometimes architects select an architectural pattern or style to shape the structural

organization of a software system, sometimes they prefer architectural tactics focusing on how to achieve a specific level of quality and finally in many cases architects make process related decisions affecting development and organizational issues in a team.

Primarily this dissertation focuses on tactical architectural decisions, therefore in the next section we only cover architectural tactics as a category of decisions made by architects.

2.2 Architectural Tactics

Architectural tactics are fundamental design decisions. They have been extensively used in a wide range of software systems, from avionic domains to e-commerce, web-based solutions and even game industries. Architectural tactics are known as reusable solutions for satisfying quality concerns such as security, performance, reliability, and so on.

Bachman et al. [7] define a tactic as

> *"A means of satisfying a quality-attribute-response measure by manipulating some aspects of a quality attribute model through architectural design decisions".*

In a simpler definition, an architect uses a set of techniques to achieve the required quality attributes, these techniques are called tactics. A tactic is a design decision that influences the achievement of a quality attribute response; tactics directly affect the system's response to some stimulus.

Len Bass [81] claims that the focus of a tactic is on a single quality attribute response and it has a single facet. Within a tactic, there is no consideration of trade-offs. He argues that this is the main difference between tactics and architectural patterns (e,g layering, pipe and filter) that trade-offs are built into architectural patterns. Patterns are multi-faceted, they are a bundle of decisions addressing multiple issues at the same time, therefore exposing trade-offs.

The Software Engineering Institute (SEI) [112] has identified a catalogue of existing architectural tactics. In this section we introduce a few categories of this catalogue including those we have used throughout the rest of this dissertation.

2.2.1 Availability Tactics

Availability tactics are critical approaches for mitigating the faults in a system and either preventing them from leading to system failure or at least bounding the effect of the faults by making system repairs possible with acceptable cost. All known approaches for maintaining system availability depend on some sort of replication, health monitoring to detect a component or system failure and finally recovery mechanisms after a failure is detected. This can be fully automated or involve intervention of system administrators. Therefore there are three strategies to handle system availability, (i) fault detection, (ii) fault recovery and finally, (iii) fault prevention.

Fault detection. The most commonly used tactics for detecting software faults are *heartbeat*, *ping-echo* and *exceptions*.

- Heartbeat (dead man timer). One component emits a heartbeat message periodically indicating that it is alive while a second component monitors the health of the heartbeat sender by listening to the heartbeat message. If the heartbeat receiver does not receive the the heartbeat message then it assumes that heartbeat sender component has failed, therefore it notifies the user/correction component.

  The heartbeat tactic can be implemented by a piggy backing mechanism. In this case the heartbeat message is carried by the communicated data.

- Ping/echo. Monitoring component emits a ping message and expects to receive back an echo within a predefined time slot from the component under scrutiny.

- Exceptions. This is one of the most commonly used programming techniques for recognizing faults. An exception is thrown when a fault is recognized. This requires an exception handler to deal with the occurred fault.

Fault recovery. The fault recovery process has two steps, preparation and recovery. In preparation the logistics, data and synchronizations for a recovery act is done while in recovery phase the actual fault recovery happens. In the following we discuss few different fault recovery tactics.

- Voting. Redundant components performing functionally equivalent tasks send their output to a voter component which is responsible for selecting the most accurate results. The architect needs to make decisions about the number of redundant processes (triple or more), and to select a voting algorithm: "majority rules", "preferred component", "NSelf Checking Programming" or some other algorithm. This architectural tactic is used to correct faulty operation of algorithms or failure of a processor (if components are deployed on different processors). Redundancy can be physical-redundancy or logical redundancy. Processes could have the identical redundancy (replication), functional redundancy or analytical redundancy. This tactic can be implemented along side N-Version Programming so that the software for each redundant component is developed by different teams, in different programming languages and is executed on dissimilar platforms or operating systems. However there is a spectrum for the dissimilarity level, and the less extreme case is to develop a single software component on dissimilar platforms.

- Active redundancy (hot restart). As the name implies, redundancy is the key factor in this architectural tactic. There are redundant components, which all respond to events in parallel. These components are running all in the same state. The system uses the response from one of these components, and the rest are discarded. In case of a fault in the primary component, the system uses results from the secondary replica. The downtime of systems using active redundancy tactic is usually very low around a few milliseconds since the secondary replica is in the same state as the primary replica and the recovery time is equivalent to the switching time from primary to secondary replica.

  Implementation of this tactic requires realtime synchronization between redundant components. This is done to ensure that all messages to any redundant component are sent to all redundant components and they are all running in the same state.

- Passive redundancy (warm restart/dual redundancy/triple redundancy). In this tactic, one component (the primary) responds to events and informs the other components (the standbys) of state updates they must make. In the case of a fault, the system has to ensure that the backup state is sufficiently fresh before resuming services. Implementation of this tactic requires periodic synchronization which is usually done by the primary component. This tactic

will result in a higher downtime compared to active redundancy but it would be cheaper and easier to implements.

- Spare. A system implemented using this tactic has a standby spare computing platform which is configured to replace many different failed components. The downtime of a system using this tactic would be a few minutes to a few hours as the system must be rebooted to the appropriate software configuration and have its state initialized when a failure occurs. Implementation of this tactic requires implementing the checkpoint mechanism to save system state to a persistent device periodically. Periodic logs of all state changes to a persistent device allows for the spare to be set to the appropriate state.

- Checkpoint/rollback. This is a tactic which uses a checkpoint to record system state during normal execution and later uses this log to recover the system to a previously safe state. The logging is done either periodically or in response to specific events.

Fault prevention. The following are some of the common fault prevention architectural tactics.

- Removal from service. The idea is to remove a component of the system from operation to prevent anticipated or predicted failures. For example, a process can be restarted to prevent the memory issues attached to that to make the whole system to go to not-responding state. The removal from services can be implemented both as manual or automated feature of the system.

- Transactions. This is a construct commonly used to achieve data integrity. A transaction is the collection of several sequential steps such that the entire collection can be undone at once. The transaction tactic is used to prevent any data from being affected if one step in a process fails. Another usage of transaction tactic is in distributed or parallel system to prevent collisions among several simultaneous threads accessing the same data. In such systems transactions are implemented through different committing protocols such as two phase commits and so on.

- Process monitor. The idea is to monitor the running processes and kill/restart them in case a fault is detected. In the implementation of this tactic there is a monitoring process which

can delete the nonperforming process and create a new instance of it, initialized to some appropriate state as in the spare tactic.

2.2.2 Performance Tactics

A wide range of performance tactics have been developed during the last decades. The main idea behind these tactics is to tune either response time, latency or throughput of a system. To deal with these performance indicators, three different categories of tactics have been developed: resource demand, resource management, and resource arbitration.

Resource Demand. User or Process requests in a software system usually causes resource demands. Performance of the system would be affected by the frequency of the requests and the resources consumed by each request. This category of performance tactic is designed to deal with the aforementioned parameters. In other words, the latency of a system can be decreased by reducing the resources required for accomplishing a task or by controlling the number of requests and minimizing it.

- Increase computational efficiency. The most common way of decreasing latency is through algorithmic improvements to performance sensitive part of a system. This includes selection of algorithms, data structures and techniques which can lead to more efficient implementation of functions.

- Reduce computational overhead. Latency can be improved by avoiding overheads among computational elements of a software such as preprocessing, intermediaries, parameters marshalling and so on. Intermediaries or communication adoptors usually are chosen in favorit of modifiability while they hinder performance of the system. This is a classic modifiability/performance tradeoff.

- Manage/control event rate by sampling. Reducing the demand is possible through partial sampling of the requests or incoming events. If there is no control over the arrival of externally generated events, queued requests can be sampled at a lower frequency, possibly resulting in the loss of requests.

- Bound execution times. Confine execution time used to respond to an event or request. This tactic is applicable in iterative or data-dependent algorithms, limiting the number of iterations or a computed threshold is a method for confining the execution times.

- Bound queue sizes. This controls the maximum number of queued requests. It is a fair sampling mechanism which controls the number of requests and consequently the resources consumed by the system.

- Asynchronous Communication. Once an operation is performed synchronously, a method is invoked, a request is sent, the results are returned, and the application resumes. In such design, one operation can be done at a time, other operations block until completion. Sometimes in order to decrease the response time or increase the throughput, it is useful to initiate a new request while another one executes. This requires asynchronous communication and method invocation, where the control returns to an application before obtaining a response.

Resource Arbitration. Resource management tactics are only applicable when there is a chance to increase resources or introduce concurrency. These tactics deal with contentions over resources by adding more resources and controlling the assignment of them. Another way to deal with resource contention is to schedule the resources such as processors, memories, buffers, networks and so on. Some common examples are:

There are various scheduling strategies. It is the responsibility of the architect to understand characteristics of each resource, recognize contextual forces of the system creating the contention over these resources and then select a proper scheduling strategy.

- First-in/First-out. FIFO is a fair scheduling strategy with treats all requests for resources as equals and satisfy them in the order they have initiated. The problem about this strategy is that, a small task can get stuck behind a very large task for long time to generate a response. Also if some of the requests have higher priority than others this scheduling strategy can be problematic as it ignores priorities.

- Fixed-priority scheduling. Fixed-priority scheduling as its name implies, considers a fixed priority for each process and assigns resources to these processes in that priority order. This

scheduling strategy results in better service for higher-priority requests while making a low-priority, but important, task wait for a long time to be serviced.

Variation of priority strategies are: *semantic based scheduling.* Len Bass defines this strategy as: "Each stream is assigned a priority statically according to some domain characteristic of the task that generates it."

Other strategies include: *deadline monotonic.* This is a static priority assignment for real-time deadlines, In this strategy the higher priority is assigned to the streams with shorter deadlines. *rate monotonic* This is a static priority assignment for periodic streams. In this strategy the higher priority is assigned to streams with shorter periods.

- round robin. Round robin is a dynamic scheduling strategy, which performs in a fair way. It orders the requests first then, at every assignment possibility, it assigns the resource to the next request in that order.
- earliest deadline first. Earliest deadline first is a dynamic scheduling strategy which assigns priorities based on the pending requests with the earliest deadline.

2.2.3 Security Tactics

Achieving security as a high level goal is dependent on a set of goals which define security characteristics in a system in terms of nonrepudiation, confidentiality, integrity, and assurance. To address these goals, architects choose tactics for resisting attacks, detecting attacks, and recovering from attacks. In this section we cover some examples of such tactics.

Resisting Attacks

- Authentication. User/Process Authentication is widely used in almost every system dealing with sensitive data or activities. This tactic ensures that a user or remote computer is actually who it is supposed to be. Username/Passwords, digital certificates, and biometric identifications are typical techniques to implement authentication.
- Message authentication: is used to provide integrity and authenticity assurances on the messages communicated between programs. Therefore a short piece of information called a message authentication code (often MAC) is used for verifying both user authenticity and integrity

of communicated messages. This would enable the integrity assurances to detect accidental and intentional message changes, while at the same time enabling authenticity assurances affirm the message's origin.

- Authorization. User/Process Authorization ensures that an authenticated user or remote computer/process has the rights to access and modify either data or services. This tactic is usually implemented through some access control patterns within a system. Access control can be by user roles/classes or by some specific policies. Therefore there are two major types of authorization. Role Based Access Control (RBAC) or Policy Based Access Control (PBAC).

- Maintain data confidentiality. Data should be protected from unauthorized access. Confidentiality is usually achieved by applying some form of encryption to data and to communication links. Encryption provides extra protection to persistently maintain data beyond that available from authorization. Communication links, on the other hand, typically do not have authorization controls. Encryption is the only protection for passing data over publicly accessible communication links. The link can be implemented by a virtual private network (VPN) or by a Secure Sockets Layer (SSL) for a Web-based link. Encryption can be symmetric (both parties use the same key) or asymmetric (public and private keys).

- Maintain integrity. Data must be delivered as it is intended. This tactic can be implemented by encoding redundant information in data, such as checksums or hash results.

- Limit exposure. An attack mainly depends on exploiting a single weakness or breach to attack all data and services on a host. The architect can design the allocation of services to hosts so that limited services are available on each host.

- Limit access. This tactic limits the ways a system can be accessed. For example Firewalls restrict access through only a set of predefined ports or sources, therefore requests from unknown sources may be a form of an attack. The negative aspect of this tactic is that, It is not always possible to limit access, especially for applications which are deployed on a cloud infrastructure or are public on the web.

- Secure Session. allows an application to only require the users to authenticate once and confirm that the user requesting a given action is the user who provided the original credentials.

This architectural decision will ensure that the authenticated users have a robust and cryptographically secure association with their session.

Detecting Attacks Attack detection tactics are mainly applied to the computer network, communication infrastructure of a system. The intrusion detection systems are the typical ways to realize this tactic. The main idea is that the behaviour of the system or network traffic is compared to patterns of normal or abnormal behaviours stored in a database. This database contains large number of historic patterns of known attacks or patterns of the normal behaviour.

- Audit Trail: is the most commonly used approach for detecting the attacker. Implementation of this tactic requires storing information about each transaction done in the system plus the identification information of that. Such audit information can be used to trace the actions of an attacker, and support system recovery. Audit trails are often attack targets themselves and therefore should be maintained in a trusted fashion.

Recovering from Attacks Architectural tactics in this category are basically either about restoring the damage or identifying an attacker (for either preventive or punitive purposes). Restoring the damage is achieved through many of the well known availability tactics. This could be implemented through introducing redundancy at multiple part of the system. Diversity is also a typical architectural tactic which is usually used along with availability tactics to increase system security. In fact, the redundant elements of a system are developed by different technology and deployed on dissimilar platforms, so that attackers can not use the same approach to bring down the replicated services used to replace the primary service under the attack.

2.3 Tactics in Action

In a preliminary study, we reviewed the design specifications for several high-assurance software systems including the Airbus A320/330/340 family, Boeing 777, Boeing 747 [11], NASA robots [65, 95, 113, 115, 116] and over twenty high-performance open-source software systems in order to determine how architectural tactics were used in actual practice [88, 89]. The study included the 16 commonly utilized tactics shown in Table 2.1, and confirmed claims by Bass [81] and Hanmer [64] that architectural tactics are commonly utilized across high-performance projects. For example, we found tactics such as *heartbeat* and *resource pooling* across the vast majority of the studied (high-performance) systems, while others, such as *active replication*, were found in a few of the more specialized systems.

TABLE 2.1: An Analysis of Tactics Across Several Performance-centric and/or Safety-critical Systems

	Fault tolerant, performance-centric software systems from SourceForge	Heartbeat	Scheduling	Authentication	Audit Trail	Resource Pool.	Active Repl.	Recovery	Passive Repl.	Authorization	Permiss. Check	CRC	Encryption	Process Monitor	Rem. Service	Fault Detection	Voting
1	RIFE: a Web application engine with support for content management.		•	•		•				•			•			•	
2	Fault-Tolerant Corba: (OMG Document ptc/2000-04-04)	•	•		•	•	•	•	•					•	•	•	•
3	CARMEN: Robot Control Software, with navigation capabilities	•										•	•				
4	Rossume: an open-source robot simulator for control and navigation.	•	•			•									•	•	
5	jworkosgi: implementation of the JMX and JMX Remote API into OSGI bundles.	•	•	•		•				•	•	•	•	•			
6	SmartFrog: Distributed Application Development Framework	•	•	•	•	•		•		•	•					•	
7	CarDamom: Real-time, distributed and fault-tolerant middleware		•	•	•	•	•	•	•					•	•	•	•
8	ACLAnalyser: Tool suit to validate, verify and debug Multi Agent Systems	•	•	•	•						•					•	
9	Jfolder: Web-based application development and management tool.	•		•	•					•							
10	Enhydra shark: XPDL and BPMN Workflow Server		•	•	•	•				•	•			•			
11	Chat3: An instant messenger.	•		•									•				
12	ACE+TAO+CIAO: Framework for high-performance, distributed, real-time systems.	•	•			•	•		•			•	•			•	•
13	Google Chromium OS:	•	•	•	•	•				•	•		•			•	
14	x4technology tools: Framework Enterprise application software.			•	•	•				•		•					
15	OpenAccountingJ: web-based Accounting/ERP system.			•	•						•						
16	Airbus Family: Flight Control System*.	•	•			•	•	•			•	•			•		•
17	Boeing 777: Primary Flight Control (PFC)*.	•	•			•	•	•				•			•		•
18	NASA CEV: Crew Exploration Vehicle using guidance-navigation* & control model.	•					•	•	•			•		•	•	•	•
19	Hadoop Framework: a development framework to support cloud computing.	•	•	•	•	•				•	•	•	•			•	
20	OfBiz: an enterprise automation and E-Commerce software.		•	•	•	•				•	•		•				

Legend: * = Tactics identified from architectural documents. In all other cases, tactics were observed directly in the code.

2.4 Architecture Erosion

In this dissertation our concern focuses on the use and preservation of architectural tactics. We therefore provide an overview for the well known problem of software architecture erosion which

alternatively sometimes called design erosion or design decay. This phenomena directly impacts architectural qualities and the intent that the architecture was designed for.

First we present a few definitions for this phenomena and its contributing causes, and we then list existing attempts to deal with this problem.

2.4.1 Definition

Perry and Wolf [100] define architectural *erosion* as 'violations of the architecture' and architectural *drift* as an 'insensitivity to the architecture' which occurs when the underlying rules are not clear to the developers and maintainers. About this issue, David Parnas [99] argues that,

> *"Programs, like people, get old. We can't prevent aging, but we can understand its causes, take steps to limit its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable. ... (We must) lose our preoccupation with the first release and focus on the long term health of our products."*-Parnas(1992)

Parnas identifies the roots of software aging in premature maintenance engineering work and residual bugs over years of software evolution. Huang et al. [69] and Grottke et al. [39] consider software erosion as an issue which helps development of software aging. In the following section, we take a detailed look at the software erosion problem specifically from an architecture perspective.

2.4.2 Causes of Erosion

Real-world industrial studies have been conducted to investigate the causes of design erosion. One of these is the work conducted by Bengtsson et.al. [127] which has identified that architecture erosion is the result of problems associated with the way software is commonly developed. These problems are:

(i) *Traceability of design decisions.* The current notations used to create software and all of its by-products such as different artifacts, lack the expressiveness needed to describe concepts used during

design. This has resulted in a situation in which it is hard for developers to track design decisions and reconstructed them from the system.

(ii)*Increasing maintenance cost.* As software evolves, it becomes more complex, and this usually results in high maintenance cost. Refactoring is not an easy task and may cause developers to make suboptimal design decisions in order to minimize the cost. This could happen for various reasons, often because the developers do not understand the architecture or because a more optimal and perfect decision would be too difficult and expensive to implement.

(iii)*Accumulation of design decisions.* Design decisions are hierarchical in nature. A high-level architectural decision is followed by many low-level decisions [89], and design decisions are accumulated and interact in a way such that revision of one would force reconsideration of all of the others. A consequence of this problem is that if programmers decide to refactor the design for any reason, they must work with a system design which is not going to be optimal.

(iv) *Iterative methods.* The goal of architecture design is to create a plan to move from requirements to implementation in a way that future change requests can be easily covered. Unfortunately, this conflicts with the iterative nature of many software development methods, especially in the agile family. These methodologies typically incorporate new requirements that may have an architectural impact during development, whereas a careful and optimized design requires knowledge about these requirements in advance.

(v)*Lack of continuous refactoring.* If refactoring is not ongoing over the life-cycle, then small design or implementation issues, architectural smells or decision inconsistencies will be accumulated, and consequently the software qualities will degrade.

Although erosion has been a commonly accepted problem of software development, there are only a few practices, techniques and methods particularly designed to mitigate this problem. The existing works in this area can be divided into two categories, *Prevention*: a strategy for using tools or practices which accept paying an upfront cost to prohibit the erosion problem, or *Repair*: postponing this concern until it is the right time to do a big refactoring to reconstruct the eroded or drifted design.[127][91]

2.4.3 Strategies to Prevent Erosion

The first step in preventing design erosion is having a more mature software development process. From a Capability Maturity Model Integration (CMMI) perspective, this would address the set of concerns identified by Bengtsson et.al. [127]. The most important factor in such a development process which can prevent design erosion is: proper documentation of architecture design, so that it is accessible to developers and gives them up-to-date knowledge about the architecture of the system. Therefore the chances that they break an architectural principle or strategic decision during code refactoring are low.

Parnas [99] emphasizes insufficient architecture design documentation, miscommunicated design principles and poor developer training as the key roots of erosion. Many of these factors can be addressed by the current artifacts, notations and practices developed in the community. Mature development processes emphasize efficient and sufficient documentation of design knowledge from requirements to design and implementation. Len Bass [81] prescribes a detailed documentation of requirements, especially qualities in form of a precise description template which is ready for rigorous analysis, as well as documenting the design architecture through different architectural views suitable for different stakeholders.

Similarly, Philippe Kruchten suggests the 4+1 View Model [77], Grady Booch [18] prescribes documentation of the architecture through multiple views and emphasizes that a single view cannot be suitable for presenting and communicating the architecture.

Beside these process related strategies, the *linkage between architecture and implementation* can provide a basis for monitoring architectural compliance at any time. Murphy et al.,[94] developed the reflexion models technique which compares a reconstructed model of the implemented architecture to a hypothetical model of the design intended by the architect. These two models are mapped by an analyst to find the deviation between intended and implemented architecture. This technique does not provide any tool support for mapping and deviation analysis. This technique is later extended in [108], the assumption is that the intended architecture exists, then repeatedly the analysts refine the implemented architecture as development progresses, and compare it against the intended architecture.

The Archium tool [25] checks runtime architectural properties. It is mainly developed to explicitly model architectural design decisions and to ensure that the implementation is aligned and consistent with the decisions. This tool considers architecture as a collection of design decisions. The idea behind this work is that capturing design decisions would prohibit design knowledge vaporization and contribute to the prevention of design erosion.

Although all of these guidelines play a role in preventing erosion, unfortunately their effect in addressing this problem has been identified as minimal. Therefore, some researchers and practitioners have adopted other strategies for dealing with architecture erosion issue.

2.4.4 Strategies to Repair Erosion

This category of approaches, is applicable once erosion has already occurred. These approaches are typically re-engineering techniques which first try to identify erosion, then reverse-engineer the system and recover the architecture from source code, and finally repair the recovered architecture to align it with the intended architecture. The existing contributions in this category can be divided into the main areas of software architecture reconstruction and refactoring. As mentioned earlier, the repair strategy would not enforce a constant cost during the software development process but instead, once dealing with erosion became necessary, it would impose a sudden cost of repairing the degraded software.

- Architecture Reconstruction Techniques

  Over the last decade, different tools and techniques have been proposed for reconstructing architecture from source code or runtime artifacts. This is mainly conducted to understand the program structure or architecture so that developers can modify the eroded system and restore it to the intended architecture, or renovate the architecture by changing it to a new optimal design.

  The Architecture Reconstruction Method (ARM) [63] is one of the first semi-automatic methods for architecture recovery from source code. This method is designed based on the pattern matching idea, to identify a set of patterns provided by the user in a reverse-engineered model of the implemented architecture. This method requires human involvement in most steps, such as specifying the patterns that might be used in the system architecture, or validating

patterns instances retrieved from searching the reverse-engineered models. Correctly detected and verified patterns, are then used to reconstruct the architecture. The actual reconstruct architecture is visualized using the Rigi visualization tool.

Reflexion models by Murphy et.al. [94] is used to map a hypothetical model of the intended architecture to the results from a static analysis of the source code. Lung et.al. [83] uses clustering techniques to refine the reflexion models and create a high level abstraction of the system structure/architecture. Classes and packages are the element of clustering techniques. Lung uses filtering to remove ad hoc elements and noises out of the abstracted models.

Mancoridis et al., [84] has created a tool called Bunch which uses dependencies between classes and generates a call graph which is later used to cluster classes and create a high level structural abstraction of the system.

Sartipi [111] has developed a software architecture recovery method based on pattern matching. This technique uses a pattern matching language called Architecture Query Language. The user needs to hypothesize the architectural patterns used in the intended architecture and describe them using the AQL. Later this description is used for searching the hypothesised pattern in the abstract graph constructed from source code.

Instead of focusing on the structure of the software, Jansen et al. [73] emphasize recovering architectural design decisions. Their approach which is called Architectural Design Decision Recovery Approach (ADDRA) is based on differences in architecture design across different versions of the system. First, detailed designs from selected versions of the implementation are recovered to generate architectural views for each version. Then the delta, which indicates the differences between these views are inspected to identify the architectural design decisions. This approach is slightly different from other approaches as it is focusing on decisions rather structure.

- Architecture Refactoring

  Architecture refactoring is widely used to reconcile implemented architecture with intended architecture. Martin Fowler has presented a set of refactoring techniques to replace and repair violation of a good design.[5] Architecture refactoring systematically restructures the software implementation code in a way that the eroded implementation stays aligned with the existing design rules, alternatively the refactoring might enforce new design rules with changing the

dependencies among packages, moving functions or alerting the partial structure of the system. Architecture refactoring is often is done to improve design fragments of software architectures which can have a negative impact on system maintainability. For example, requirements change and this might results in the adoption of a design solution which is inappropriate for that context, or the new solution might result in undesirable behaviour. In such situation, refactoring the architecture is necessary to remove the issues known as architectural smells and prevent their accumulation which may end in design erosion. Many researchers also have conducted studies on identification and categorization of various architectural smells, [53][54] architectural antipatterns [22], and refactoring of a design [47][5]. A number of different tools and IDEs have been developed to support code refactoring; unfortunately, none of them emphasize the architecture-level concerns. Therefore it remains the developers' responsibility to take care of these issues.

2.5 Summary

Our focus in this dissertation is on architectural tactics. Like other architectural concerns they have a tendency to erode over time. Given their importance for realizing critical system qualities it is important to preserve them. This chapter provided a summary on several different types of architectural tactic. Furthermore it discussed the problem of architectural erosion and articulated the current notions used to address this important problem.

"The biggest difference between time and space is that you can't reuse time;"

Merrick Furst

Chapter 3

Traceability Fundamentals

This chapter provides a quick overview of traceability fundamentals and introduces the essential traceability terminology and concepts used in this dissertation.

3.0.1 Definition of Software Traceability

The COEST [33] defines Requirements Traceability as,

> "the ability to interrelate any uniquely identifiable software engineering artifact to any other, maintain required links over time, and use the resulting network to answer questions of both the software product and its development process".

The *source artifact* is the artifact from which a trace originates, and the *target artifact* is the artifact at the destination of a trace. Software traceability is the ability to relate the concepts and data within a software artifact (source artifact) to another software artifact (target artifact). This can be done by explicitly connecting the artifacts together (e.g establishing a trace links) or implicitly providing this connection (e.g. tagging the artifacts).

The IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990) [4] provides a more general definition of traceability:

> "the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another".

Explicit traceability results in the creation of *trace links* which is a specified association between the source artifact and the target artifact. A trace link can have tag indicating the semantics of the link, and by default the direction of a trace link is from source artifact to target artifact, however it is possible to have inverse or bi-directional trace link.

Software traceability has been identified by many organizations such as the Federal Drug Administration (FDA) [3] , Federal Aviation Administration(FAA) [6] and Department of Defence (DOD) [49] as a mandatory software engineering practice for safety critical software projects in order to accomplish compliance verification, impact analysis, regression test selection, safety-case construction, requirements allocation and coverage analysis.

3.0.2 Traceability information model (TIM)

The traceability information model (TIM) is a graph style model which defines the trace artifact types, the trace link types and the permissible trace relationships on a project, in order to address anticipated traceability-related queries and traceability-enabled activities and tasks [58].

Best practices for software traceability dictate that each particular project needs a TIM expressing what should be traced in the project. A TIM can capture additional information such as semantics of trace links, cardinality and link direction.

Creating a TIM can help to ensure that traceability is established strategically for a project. Ramesh [104] has developed a set of well known but generic traceability information models through systematic analysis of the traceability problems. (more details on [104])

3.0.3 Tracing and Related Concepts

The operation of establishing trace links between different artifacts or using those trace links is called *tracing*. Tracing like any other computer aided activity, can occur in three different forms: manual, automated and semi-automated [57].

In *manual tracing*, a human establishes and uses the traceability links. Although there should be some basic tool support such as drag and drop features to decrease most of tracing burden.

Conversely in *automated tracing*, the trace links are established via automated techniques, methods and tools. Finally, in *semi-automated* tracing the trace links are established via a combination of automated techniques, methods and tools and human activities.

In all of these cases, the trace links can be established either through *traceability creation* by associating two (or more) artifacts, *trace capture* by creating the trace links concurrently with the creation of the artifacts, or *trace reconstruction* by establishment of trace links after the generation or manipulation of associated artifacts.

Forward traceability is tracing in the same order as the artifacts appear in a developmental path, which is not necessarily a chronological path, such as forward from quality requirements through design decisions to source code.

Backward traceability is tracing in the antecedent steps in a developmental path, again not necessarily a chronological path, such as backward from code through design decisions to quality requirements.

Trace granularity indicates the level of detail at which a trace is recorded and performed. The granularity of a trace is defined by the granularity of the source artifact and the target artifact. For example, at a coarse-granularity, an architectural decision is traced to a subsystem or component, while the same decision at fine granularity could be traced into two different methods in a Java class.

3.0.4 Automated Traceability

Several IR methods have been developed to automate the creation and recovery of the trace links. These approaches work based on the similarity between the words in the text contained software artefacts. A trace link is established between two artifacts if they have high textual similarity.

Typically utilizing IR methods contains the following steps:

1. pre-processing of the documents to extract the vector of words.

2. corpus indexing with an IR method.

3. ranking the candidate links.

There are several strategies to pre-process a textual document. For example *text normalization* can be used to remove white spaces and non-textual characters from the text. *stop word removal* can be used to remove stop words or common words (i.e., articles, adverbs, etc) that are not useful to capture the semantics of each artefact. Furthermore, many times *Stemming* is used for reducing inflected or derived words to their stem or root form.

After preprocessing the documents several different IR methods can be used to create the trace links. Based on a survey conducted on the available research papers indicated that probabilistic models [10, 36], VSM [16], and LSI [101] are the three most frequently used IR methods for traceability recovery.

Jane Cleland-Huang et.al [36] have used the probabilistic model based on conditioned probability to recover links between requirements and UML diagrams, while others have utilized similar probabilistic models to recover the trace links between requirements and source code [9, 10], and links among requirements and regulatory codes. [31].

Several researchers have used the VSM to recover traceability links among requirements [66], requirements and source code [9], [10], [85], test cases and source code [44], and defect reports and source code [133].

Despite the wide usage of VSM, there is a major criticism on VSM that it does not take into account relations between terms [45]. For instance, having a term like “programmer” in one document and another term like “developer” in another document does not contribute to the similarity measure between these two documents. Therefore LSI [45] was developed to overcome this problem by taking into account the synonymy problems, which occur with the VSM model.

LSI addresses this problem by explicitly taking into account the dependencies between terms and documents. It assumes the existence of a “latent structure” in word usage and uses statistical techniques to estimate this latent structure. For example, both “programmer” and “developer” are likely to co-occur in different documents with related terms, such as “codes”, “programs”, etc.

3.0.5 Event-Based Traceability

Event-based traceability (EBT) developed by Cleland-Huang et.al.[29] provides an infrastructure for using trace links to answer change impact analysis queries. It also supports long-term evolutionary change so that the traceability scheme can be maintained.

In Event-Based Traceability (EBT), source artifacts can be considered as publishers of change events and target artifacts as subscribers. When a source artifact goes through some changes, all subscribers are notified of the change. The event-notification architecture enables the EBT traceability scheme to handle change robustly.

3.1 Tracing Architectural Concerns

Non-functional requirements, describing quality concerns such as performance, reliability, availability, and security, often exhibit complex interdependencies and trade-offs [80][43] and have broad-reaching impacts across the architectural design and implementation of a software intensive system. Non-functional requirements are often realized in the design and implementation code as architectural tactics.

The cross cutting nature of NFRs, creates significant challenges for tracing them, resulting in the proliferation of traceability links. As a result, tracing NFRs can be expensive, and unfortunately many organizations do not even attempt to trace them; however this means that change requests during software maintenance are often implemented with very little understanding of how system qualities are affected by the change. This section summarizes existing practices and techniques developed for tracing NFRs.

3.1.1 Software Architecture Practices that capture NFR traces

In practice, many architectural assessment and project scoping techniques *implicitly* trace NFRs into architectural designs [62][103][76][75]. Implicitly means that the relationships between NFRs and design decisions are created or embedded in the artifacts without establishing a traceability

matrix. The benefits of building implicit traceability on top of such methods is that project stakeholders realize immediate benefits from their traceability efforts. The disadvantage is that NFR traceability links are often embedded in documentation that is specific to a given activity, and it is therefore difficult to extract and use those links to support other unrelated activities. For example, NFR trace links which are created and documented within an architectural analysis document, will likely be available for future architectural analysis activities, but will not be readily available for programmers who may need to understand how a low level code modification impacts an architectural decision. This section, provides an overview of four software engineering methods and activities that incorporate the creation and utilization of NFR traceability links. These methods and practices are the Architectural Tradeoff Analysis Method (ATAM), Architectural Documentation, Enterprise Architectural Frameworks, and management of architectural knowledge.

- **Architecture Tradeoff Analysis Method (ATAM)**

The Architecture Tradeoff Analysis Method (ATAM) is a qualitative approach to risk and trade-off analysis of an architecture with respect to a set of clearly articulated quality scenarios [81][76][75]. The evaluation process starts with a presentation of the business drivers, including a high level description of quality attributes such as security, safety, or reliability. This is followed by a short presentation of candidate architectures. Quality scenarios are then generated and prioritized, and architectural solutions addressing those qualities are identified and documented. As a result of these steps, the prioritized NFRs are mapped onto their corresponding architectural decision and the design fragment in which the architectural solution is implemented. and the NFRs are then evaluated to determine how well they address the specified quality attributes.

ATAM implicitly documents traceability relationships among quality scenarios and the architectural elements in which they are realized. These mappings create a de facto traceability matrix, documenting relationships between quality concerns, tactical architectural decisions, and lower level design solutions. Unfortunately, as previously noted, this information is not easily accessible for any purpose other than architectural analysis. However, Mirakhorli et al [90] have partially addressed this problem by developing a utility for extracting traceability information from ATAM documents and using it to construct a more traditional traceability matrix.

- **Architecture Documentation Methods**

Architectural documentation approaches such as Views-and-Beyond [81], Siemens S4V [68], and RUP 4+1 [77] provide guidelines and a template for documenting architectural solutions across multiple views. Each view depicts a coherent set of architectural elements from a specific perspective such as hardware resources, runtime behavior, or data usage, and is presented visually with a supporting catalog describing the behavior and property of each element, its interfaces, and the qualities associated with each interface. Architectural decisions and rationales associated with each view are also documented.

The catalog of elements implicitly captures traceability relationships among architectural elements and the quality concerns exposed by a component or its interfaces.

- **Enterprise Architectural Frameworks**

An enterprise architectural framework provides a mechanism for describing and communicating architectural concerns, for comparing different architectural solutions, and for helping to ensure the integrity and completeness of a solution. Several architectural frameworks, including the Command, Control, Computers, Communication, Intelligence, Surveillance, and Reconnaissance (C4ISR) framework [24], have directly addressed issues of tracing quality concerns. C4ISR was developed by the U.S. Department of Defense (DoD) to improve the operational capabilities of warrior systems across defense agencies. The C4ISR framework encompasses three different architectural views. The Operational View (OV) artifacts define operational elements, activities and tasks, as well as the information exchange needed to accomplish an operation. The System View (SV) artifacts describe the physical systems, software services and interconnections needed to support operations. Finally the Technical View (TV) defines technical standards, implementation conventions, rules and criteria governing interaction and interdependences of system parts. C4ISR utilizes traceability in several different ways. For example the System Interface Description is used to map supporting security and communication requirements to system interfaces, while the Operational Information Exchange Matrix (OV-3) is used to describe operational node connectivity characteristics such as Throughput, Security, Timeliness (e.g., 10/minute), and Required Interoperability Level. In these cases, the traceability matrices are used to map qualities to software elements; however the traces are relatively high level and do not provide detailed mappings from quality concerns to subsystems.

- **Knowledge Management Tools**

Software architectural knowledge management tools provide support for documenting architecturally significant requirements, the architectural decisions that were made to satisfy those requirements, and the rationale behind those decisions [12]. Documenting architectural knowledge helps developers and architects maintain existing systems and can also be used to improve the architectural design of future systems. Tyree and Akerman [123] proposed a taxonomy of items needed to effectively document design rationales including issues, decisions, assumptions, arguments, implications, related decisions, related requirements, related artifacts, related principles, and notes. Other researchers, such as Kruchten [79] and Burge [23], have proposed similar ontology to document architectural decisions. All of these works assume the underlying use of traceability links to relate architectural decisions to external artifacts such as requirements, design documents, and architectural assessments. Several tools have been developed to capture and re-use architectural knowledge. Although the primary focus of these tools is on architectural knowledge, the organization of that knowledge relies upon user-created traceability links.

Most architectural management tools, such as Process-based Architecture Knowledge Management Environment (PAKME) [13], Archium [72] and Architecture Design Decision Support System (ADDSS)[25] , help architects to create traceability links between knowledge related items, such as requirements and design decisions, and external documents. However, the tools we evaluated support only relatively coarse-grained traceability between documents and do not support finer grained traceability between NFRs and specific design or code elements in which architectural decisions are realized. Furthermore, the tools have not yet been integrated with architectural modeling tools [117], which further limits their ability to support NFR traceability to critical elements of the architecture.

3.1.2 Custom Processes and Techniques for Tracing NFRs

In addition to software architecture practices which utilize NFR traceability to support their prime objectives, there are several other techniques, some of which are designed specifically for creating and maintaining NFR traces. In this section we describe four techniques including use of UML Profiles, Goal-Centric Traceability, Tracing through Design Patterns, and Decision-Centric Traceability. The benefit of these approaches is that they provide higher degrees of automation for using

and understanding traceability links, and in some cases are designed specifically with maintainability in mind; however unlike the methods described in the previous section, these approaches are constrained to either specific modeling environments or development practices, and do not necessarily return immediate benefits to the trace creators. This makes their consistent, project-wide use unlikely.

• **Techniques that Embed Traceability Links into UML** The Unified Modeling Language (UML) is used to visualize, specify, and construct elements of an object-oriented system. It models boundaries and interactions between the system and its users, the communication between objects, the state of those objects, the static structure of the system, and the system's physical architecture[109]. Standard UML can be customized for a particular domain through the use of UML Profiles, which allow the semantics of standard UML elements to be refined via stereotypes, tags, and the object constraint language (OCL) [130]. For example, a «trace» stereotype could be created and associated with a dependency link to depict a traceability relationship.

Several researchers have developed UML profiles for supporting traceability of NFRs. For example, Salazar-Zrate [110] modeled NFRs and related them to functional elements through use of a «NFR Behaviour» stereotype, and then described behavioral attributes using OCL.

The Architecture Rationale and Element Linkage (AREL)[118] approach provides two new UML profiles for modeling Architectural Entities (AE) and Architectural Rationales (AR). These profiles, and an associated tool, allow architects to visualize AEs and their related ARs. AEs can represent functional requirements, non-functional requirements, components, processors, or text documents; while ARs describe quantitative rationales such as the costs, benefits, and risks associated with architectural decisions, and the qualitative rationales which document the issues, arguments, alternatives, and trade-offs associated with a design decision. An AREL model is represented as an acyclic graph, in which causal dependencies between design rationales and design objects can be traversed in order to extract traceability links.

In related work, Zhu et al [135] have proposed two UML profiles for modeling architectural decisions and NFRs. One of the limitations of embedding traceability links into UML diagrams is the fact that trace links are limited to individual models. Cysneiro et al. addressed this limitation by developing a Language Extended Lexicon (LEL) that facilitated the tracing of goals across multiple UML

diagrams. Their approach embedded controlled keywords from the LEL into goals and elements of the UML models [42].

UML approaches enable traceability relationships to be depicted within the design model, but suffer from scalability problems which make even medium-sized models difficult to create and understand. Furthermore, many UML profile approaches are limited to tracing structural elements and exclude traces to a broader set of models such as deployment or implementation models. Both of these issues are major short-comings, as both scalability and heterogeneity were identified through our study of safety and performance-critical systems as fundamental requirements for tracing NFRs. Despite these issues, the idea of incorporating traceability into UML models is quite appealing, simply because UML models are a natural part of many software development projects. However, it should be noted that such approaches focus on the notation of the trace links and provide very limited guidance for how and where to establish useful and effective links for tracing NFRs.

- **Aspect Oriented Approach:**

Aspect Oriented Requirements Engineering (AORE) approaches focus on identifying cross-cutting concerns, many of which are architectural in nature [61, 106]. As a precursor to Aspect Oriented Programming (AOP), AORE's primary purpose is to identify candidate cross-cutting concerns, some of which will later be recognized as aspects and implemented as such in the final code. The concepts of AORE provide an enticing framework for tracing NFRs, as many early aspects do in fact represent specific quality requirements. For this reason, several researchers have explored ideas of using early aspects to trace NFRs [96, 105, 120]

The first approach, referred to as "Aspect-oriented development model with traceability mechanisms" [96] facilitates the separation, composition and traceability of cross-cutting concerns (both functional and nonfunctional). This approach includes a dynamic view in which cross-cutting concerns are traced to use-cases and scenarios, and a static view in which they are traced to conceptual classes. In related work, Tekinerdogan [120] developed a concerns traceability meta-model (CTM) for tracing concerns throughout the life cycle. The meta-model provides support for bidirectional traceability between concerns in the requirements and design, and for traces between concerns and other artifacts.

• **Goal Centric Traceability:** Goal-Centric Traceability (GCT) provides traceability support for managing and maintaining NFRs and their related quality concerns over the long-term life of a software intensive system [34]. As its name suggests, GCT is a goal-oriented approach which assumes that quality concerns are modeled in a goal hierarchy such as the NFR framework [80], i* [134], Tropos [26], or an ATAM utility tree [81]. GCT also assumes that during the initial analysis, design, and implementation of the software system, a number of different models are developed to evaluate the quality of the design. These might include ATAM scenarios for evaluating how well a design satisfies critical use cases, a Software Performance Execution (SPE) graph to evaluate response time goals, a system execution graph to measure throughput and latency [128], an attack graph [118] to evaluate security attributes, usability metrics to evaluate a graphical user interface, or an executable test-case to evaluate functionality that is needed to satisfy quality goal. GCT refers to these kinds of models as Quality Assessment Models (QAMs) [34]. The GCT framework, includes: (i) a goal model that captures stakeholders' quality concerns and their trade-offs, (ii) a set of QAMs that have been designed to evaluate the extent to which the architecture satisfies the stated quality goals, (iii) a traceability infrastructure that is used to link QAMs to goals, (iv) GCT algorithms that manage the automated impact analysis and propagation of change across the goal hierarchy, and finally (v) an impact report which describes the potential impact of a change on the overall quality goals.

GCT supports two specific traceability tasks. The first involves identifying the initial impact of a change upon the GCT model. The second traceability task is triggered once an initial impact point is discovered. This second task is internal to the GCT model, and utilizes the internal structure of the goal model, the executable traceability links between specific goals and QAMs, and the GCT propagation algorithms. In GCT, an executable trace is defined as a trace which carries sufficient semantics to be processed automatically, so that the QAM can be parameterized and re-executed, and output values are returned to the GCT model for evaluation.

The primary advantage of GCT is that it provides support for maintaining quality concerns over the long-term by making use of QAMs that were already created during the initial development phase; however GCT is only viable if tool support is available to automate the process, and if QAMs, such as simulation models, are created as an integral component of the development process. GCT is

therefore best deployed for only a critical set of goals, for which executable QAMs are available, and as such cannot be seen as a holistic solution for tracing all NFRs.

• **Design Pattern-Based Approaches:** A design pattern represents a reusable solution to a commonly occurring problem in software design [51]. In addition to solving a specific functional problem, design patterns often address a specific class of quality concerns such as maintainability, flexibility, or portability. Gross and Yu [59] and Cleland-Huang [35] proposed tracing NFRs to software designs through the use of existing design patterns as intermediaries. This technique supports traceability for any NFR that can be implemented as a design pattern.

3.2 Summary

This chapter described the fundamental traceability concepts used in this dissertation. Furthermore it summarized the key approaches previously used for tracing quality concerns. The work described in the following chapters relies on these concepts and extends the existing work already published in the area of software architecture traceability.

Part II

Creating Architecture Traceability

"I remember my mother's prayers and they have always followed me. They have clung to me all my life."

Abraham Lincoln

Chapter 4

Decision Centric Traceability

The architecture of a system and its qualities can be maintained with the support of traceability methods and a related change-impact analysis infrastructure. This infrastructure could help the developers to fully understand the impact of design or implementation changes they make on architecturally significant requirements and quality concerns. Unfortunately, the current state of software traceability methods does not provide this level of support. Existing traceability meta-models, solutions or techniques do not provide specific guidelines for how traceability links should be established between quality concerns and software design and implementation artifacts. Furthermore, current methods tend to result in a proliferation of traceability links and create a situation in which traceability is difficult and costly to maintain, and in which links degenerate into an inaccurate and non-useful state.

Contribution: In this chapter, the important challenges of architectural traceability have b identified and summarized through an extensive study of architectural decisions in highly dependable and complex avionic systems. The conducted study involved reviewing the specifications of several high-assurance software systems including the Airbus A320/330/340 family, Boeing 777, Boeing 7J7 [11, 113] , NASA robots [70, 95], NASA Crew Exploration Vehicles [65, 115, 116] and also implemented code of performance-centric systems such as Google Chromium OS, Hadoop Framework etc. Furthermore these traceability challenges have drew the novel concept of Tactic Traceability Patterns (tTPs). tTPs reduce the cost and effort of traceability through providing a set of re-usable traceability links.

4.1 Introduction

The proliferation of traceability links is one of the main issue of using traceability practices. This issue is visible through series of Ramesh's [104] traceability meta-models that were produced from a study of traceability practices in industry. The Traceability Information Model (TIM) depicted in Figure 4.1, is derived from these meta-models and shows significant redundancy of traceability paths for establishing relationships between issues, conflicts, alternative options, arguments, rationales, assumptions, requirements, and design decisions.

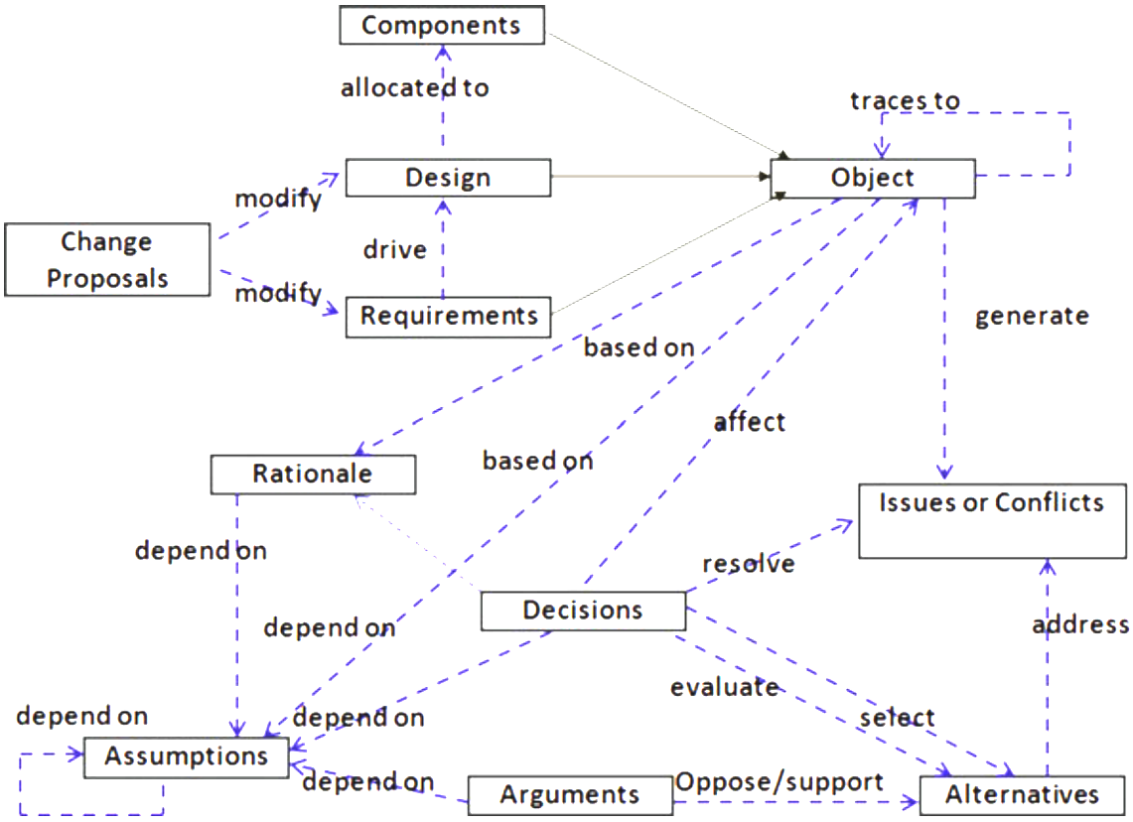


FIGURE 4.1: Components from Ramesh's Metamodel - Rationale SubModel [104]

For example decisions can be traced directly to requirements, or they can be traced indirectly through either rationales or through issues and conflicts. There is a lack of any guidance to inform architects as to the best way to establish traceability. Moreover, various studies have significantly emphasized the need to simplify the creation and use of traceability [8].

Proliferation of traceability links is exacerbated when tracing architectural concerns which describe quality attributes such as performance, security, reliability, and maintainability. Such concerns

often have a cross cutting nature, and therefore exhibit a broad reaching impact across the system. These concerns are often realized through components and behaviors that are visible in a variety of architectural and implementation views at very different abstraction levels. Here the main criticism to standard traceability meta-models arises as they do not begin to address this degree of complexity and cannot tame it.

Before prescribing specific traceability guidelines for tracing quality concerns, it is necessary to understand the issues that must drive any effective architecture traceability solution. Therefore, we first report the results of an extensive study of architectural decisions that we conducted in highly dependable and complex avionic systems. The conducted study involved reviewing the specifications of several high-assurance software systems including the Airbus A320/330/340 family, Boeing 777, Boeing 7J7 [11, 113] , NASA robots [70, 95] see AppendixA.1, NASA Crew Exploration Vehicles [65, 115, 116] and also implemented code of performance-centric systems such as Google Chromium OS, Hadoop Framework and so on. For each of the systems studied, we identified critical quality goals, architecturally significant requirements, architectural decisions, tactics, patterns, design solutions, and views and models in which each of these techniques were visible. This study provided the foundations and motivation for a new decision-centric traceability approach that includes a meta-model describing the required traceability links, and a strategic process for applying the meta-model.

4.2 Identified Challenges

As a result of this study we observed seven issues that significantly can influence the proposed traceability approach. Each of these is discussed below:

• **Decisions are Hierarchical in Nature.** It is quite common that architectural decisions have a hierarchical nature. Architects usually start with a high level decision, then they adopt different low level supporting decisions to tune the effectiveness of their initial high level decision. Although both high and low level decisions are often traceable back to individual driving user or system level requirements, it is often only the lower-level decisions that are traceable forward to the architectural views or source code. For example, in all of the avionics cases studied, the reliability requirement was at least partially achieved through a decision to deploy redundant components; however this high level decision was realized through a variety of different sub-decisions. The airbus architecture

utilizes logical redundancy through use of multi-process threading, while the Boeing architecture deploys components on different processors. In the Boeing 777, the Primary Flight Control (PFC) system includes three replicated software modules deployed on separate processors [11]. At runtime, each of the PFCs receives the same input values from which they independently compute output values, and the results are evaluated using a "majority" voting algorithm.

In contrast, the Airbus flight control system uses a "2-Self Checking Programming" as the voting mechanism. This includes two software units for command and monitoring, which also receive identical inputs. The units are located on a single processor but run on separate operating system processes. The results generated by the command unit are compared to those of the monitoring unit and in the case that the results do not match, airplane control is switched to another computer. Another architectural decision that we observed in Airbus case studies, involved the use of N-Version Programming to mitigate against the possibility of duplicating design errors across redundant components. In this case, sub-decisions involved selecting programming languages, making team assignments, adopting different algorithmic solutions, and using different hardware and software platforms.

These examples demonstrate that high-level architectural decisions are often associated with a fairly extensive set of subsequent lower-level decisions which impose constraints on the behavior, structure, and deployment of the system, and which work synergistically to support and shape the higher level decision. Tracing the high-level decision therefore translates into first understanding how a high-level decision is realized through lower level decisions, and secondly understanding how to effectively trace those lower level decisions into the architectural design and implementation.

• **Visibility of Architectural Decisions.** Different architectural decisions are visible in different views, therefore traceability links must be established across a wide variety of architectural views. From observing the various architectural decisions in both the avionics systems and in non-mission critical system such as Chrome, Hadoop and OfBiz„ it is evident that different decisions have very different scopes of impact. While some decisions, such as Single Sign in (SSN), may deal with the structure of the system and lead to creation of sub-systems, layers or specific components, other decisions, such as fault recovery or performance decisions, affect how elements interact, perform their responsibilities or appear at runtime, and are more likely to be visible in behavioral models and runtime views. Furthermore, a layered view may show tactical decisions concerning the system's

portability, while a deployment view may depict decisions concerning the system's performance and reliability. Some decisions are visible in multiple views, for example a decision might be traced to both architectural models (e.g., a class, an interface, a process or thread, a package or subsystem etc.) as well as implementation artifacts. An effective traceability scheme therefore needs the ability to trace quality concerns to a wide variety of architectural and implementation views.

• **Granularity of Architectural Impact.** Many architectural decisions are characterized by a set of roles and constraints. For example, an availability tactic such as heart-beat is traced at a coarse-grained level to a component that emits a heartbeat message and another that listens for and monitors those messages and reacts accordingly. However, it may also be traced to one or more variables controlling the heart-beat rate. To preserve system reliability we need to trace the heart-beat decision to both the component level and to the variable level. Traceability links must therefore be created and maintained at various levels of granularity.

• **Tacit Architectural Knowledge.** Sometimes architects make tactical decisions but these decisions are not explicit. In fact these decisions represent tacit knowledge which is rarely articulated and, by definition, never documented, they exist but in form of tacit knowledge in the head of architects and developers. To prevent architectural erosion, tacit knowledge must be articulated as a design decision, even if it is not traceable into any specific architectural view. For example, an architect might decide that concurrent threads should not have direct write access to shared data. Because such decisions are implicit, our study did not reveal them explicitly. Nevertheless, they are included in this discussion, because of their importance to the traceability issue. If traceability is to be used to prevent architectural erosion, tacit decisions must be explicitly articulated, documented, and traced.

• **Architectural Trade-offs.** Design decisions exhibit tradeoffs and interdependencies. It is necessary to capture such interdependencies so that the impact of a change to one decision is understood across the broader context of other architectural decisions. Our study identified numerous examples of potential design trade-offs. For example, redundancy requirements in avionics systems, trade-off against weight requirements and performance requirements, and similarly security requirements in Chrome trade-off against performance. Relationships between decisions, including both positive contributions and negative trade-offs need to be explicitly modeled as traceability links.

• **Rich Semantics.** Balasubramaniam's prior study on traceability highlighted the need for traceability links to be semantically typed [5]. This is especially important in tracing architectural concerns because of the various roles played by different components in realizing architectural decisions. For example, a given component might provide diversity for an N-Version design strategy, while another component might be responsible for coordinating voting tactics. Semantically typing each traceability link provides enhanced support for making sense of the traceability links, in a way that the links convey more conceptual informations and can be used for better support of important tasks such as architectural preservation.

• **Minimalistic Strategy.** Complex high-assurance and high-performance systems are rich with design decisions. Given the known problems of creating, maintaining, and using traceability links, it is important to develop a minimalistic traceability strategy that removes redundancy, while retaining only those traceability links needed to support critical software engineering tasks such as impact analysis and architectural preservation.

4.3 Decision-Centric Traceability Meta-Model

As a result of the above study and after careful considerations of all the observations from these systems, we developed a decision-centric traceability meta-model for capturing architectural design decisions and creating traceability links between architectural components and the non-functional requirements and concerns they are designed to satisfy. The model is depicted in Figure 4.2, and shows that traces are established from quality concerns, through architectural decisions to architectural elements visible across various architectural views. In this meta-model the trace redundancy is removed, while the expressivity required for a traceability model is maintained. Figure 4.3 illustrates how the model might be instantiated for a redundancy decision that involves voting and N-Version programming.

Establishing all traces through architectural decisions, and disallowing traces from quality concerns directly to the architectural design, addresses all of the issues identified through our conducted studies. As depicted in Figures 4.3, decisions can be structured hierarchically to capture the way in which sub-decisions shape and help to achieve higher level decisions. Traceability links can then be created at any granularity level through the hierarchy of decisions. The proposed meta-model

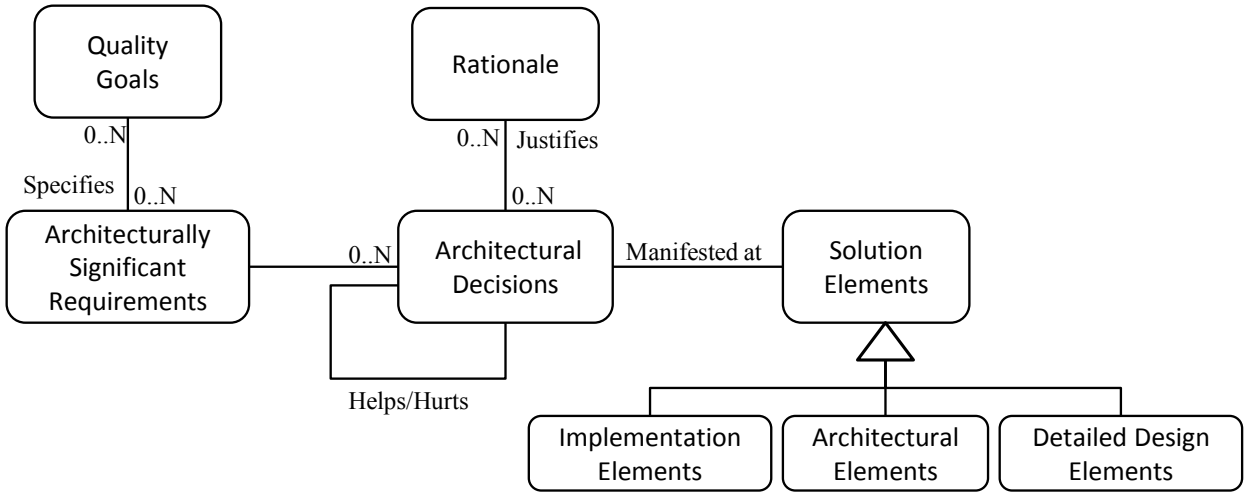


FIGURE 4.2: Decision-Centric Traceability (DCT) Meta-Model

promotes capturing traceability links around design decisions and therefore it supports visualization of design knowledge. This increases the program comprehension and provides enhanced support for helping trace users to understand the impact of a decision across multiple architectural views. Since architectural decisions are frequently realized through the use of standard tactics, architectural patterns, or constraints, many of which include recognized roles; traceability links can be semantically typed to depict specific roles played by the architectural component in realizing a specific decision. For example, in Figure 4.3, traceability links to architectural components contributing to the redundancy tactic are semantically typed as "Coordinates", "Assigned to", "Provides Diversity", which immediately conveys the role of the component in realizing the decision.

The proposed meta-model suggests tracing the requirements through design decisions into design components and implementation modules. We extended the notion of the meta-model for each specific architectural tactics, and proposed an augmented model in which, it is clear where to create traceability links in order balance the costs versus benefits of tracing architectural concerns. [46].

This extension results in the Tactic Traceability Pattern (tTP), designed to help architects establish strategic traceability links which can be used during the maintenance phase to visualize underlying architectural tactics and tradeoffs, and to keep maintainers fully informed of underlying decisions and rationales. The proposed tTPs explicitly differentiate between reusable traceability links i.e. those links internal to the tactic, which can be used across multiple projects, versus project specific

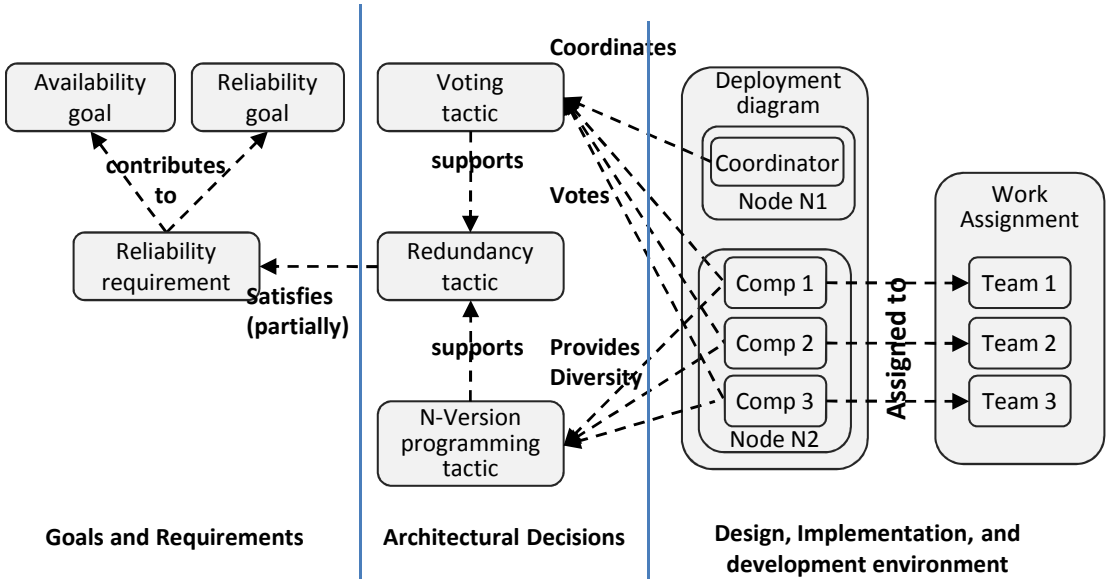


FIGURE 4.3: An example of tracing and visualizing the redundancy tactic using DCT meta-model

traceability links established as mappings from concrete elements of the architectural design to components of the tTPs.

In this part of work, we are interested to examine the following research questions,

- ***RQ1.*** *Does using tTPs potentially reduce the cost and effort of establishing and maintaining traceability links?*

- ***RQ2.*** *How useful are tTPs for notifying the developers of potential erosion through architecture-change impact analysis?*

All the tTPs described in section of the dissertation, were identified through observing the use of architectural tactics across several high-assurance software systems including the Airbus A320/330/340 family, Boeing 777, Boeing 7J7 [114], NASA robots [70, 95], NASA Crew Exploration Vehicles [65, 113, 115, 116], Google Chromium OS[2] and many other and over twenty high-performance open-source software systems. We conducted an study which included the 16 commonly utilized tactics shown in Table 4.1.

TABLE 4.1: An Analysis of Tactics Across Several Performance-centric and/or Safety-critical Systems

	Fault tolerant, performance-centric software systems from SourceForge	Heartbeat	Scheduling	Authentication	Audit Trail	Resource Pool.	Active Repl.	Recovery	Passive Repl.	Authorization	Permiss. Check	CRC	Encryption	Process Monitor	Rem. Service	Fault Detection	Voting
1	RIFE: a Web application engine with support for content management.		•	•		•				•			•			•	
2	Fault-Tolerant Corba: (OMG Document ptc/2000-04-04)	•	•		•	•	•	•	•					•	•	•	•
3	CARMEN: Robot Control Software, with navigation capabilities	•										•	•				
4	Rossume: an open-source robot simulator for control and navigation.	•	•			•									•	•	
5	jworkosgi: implementation of the JMX and JMX Remote API into OSGI bundles.	•	•	•		•				•	•	•	•	•			
6	SmartFrog: Distributed Application Development Framework	•	•	•	•	•		•		•	•					•	
7	CarDamom: Real-time, distributed and fault-tolerant middleware		•	•	•	•	•	•	•					•	•	•	•
8	ACLAnalyser: Tool suit to validate, verify and debug Multi Agent Systems	•	•	•	•						•					•	
9	Jfolder: Web-based application development and management tool.	•		•	•					•							
10	Enhydra shark: XPDL and BPMN Workflow Server		•	•	•	•				•	•			•			
11	Chat3: An instant messenger.	•		•									•				
12	ACE+TAO+CIAO: Framework for high-performance, distributed, real-time systems.	•	•			•	•		•			•	•			•	•
13	Google Chromium OS:	•	•	•	•	•				•	•		•			•	
14	x4technology tools: Framework Enterprise application software.			•	•	•				•		•					
15	OpenAccountingJ: web-based Accounting/ERP system.			•	•						•						
16	Airbus Family: Flight Control System*.	•	•			•	•	•			•	•			•		•
17	Boeing 777: Primary Flight Control (PFC)*.	•	•			•	•	•				•			•		•
18	NASA CEV: Crew Exploration Vehicle using guidance-navigation* & control model.	•					•	•	•			•		•	•	•	•
19	Hadoop Framework: a development framework to support cloud computing.	•	•	•	•	•				•	•	•	•			•	
20	OfBiz: an enterprise automation and E-Commerce software.		•	•	•	•				•	•		•				

Legend: * = Tactics identified from architectural documents. In all other cases, tactics were observed directly in the code.

4.4 Tactic Traceability Patterns

A tTP provides the required infrastructure for tracing a specific architectural tactic into the design and implementation. Each tTP is centered around a tactic and defines both backward traceability to the driving requirements and rationales of the tactic, and forward traceability to the architectural elements in which it is realized. Moreover, a tTP defines the internal structure of a tactic in terms of its primary roles and parameters, also relationships between the roles and proxy elements which are used to map architectural elements in design models and code (i.e. concrete classes, methods, variables, files etc in a project) to the tTP. which are used to map architectural elements in design models and code to the tTP. Forward traceability is done by using these proxies. Traceability links between tactics and architecture are therefore established as mappings between a proxy and concrete design or implementation element. Backward traces include links to goals, quality requirements and rationales associated with the tactic, however if necessary, these can be customized for a specific project.

tTPs primarily lead to effective traceability when building families of systems which implement similar tactics, or when using very common tactics that tend to be implemented in similar ways across different products.

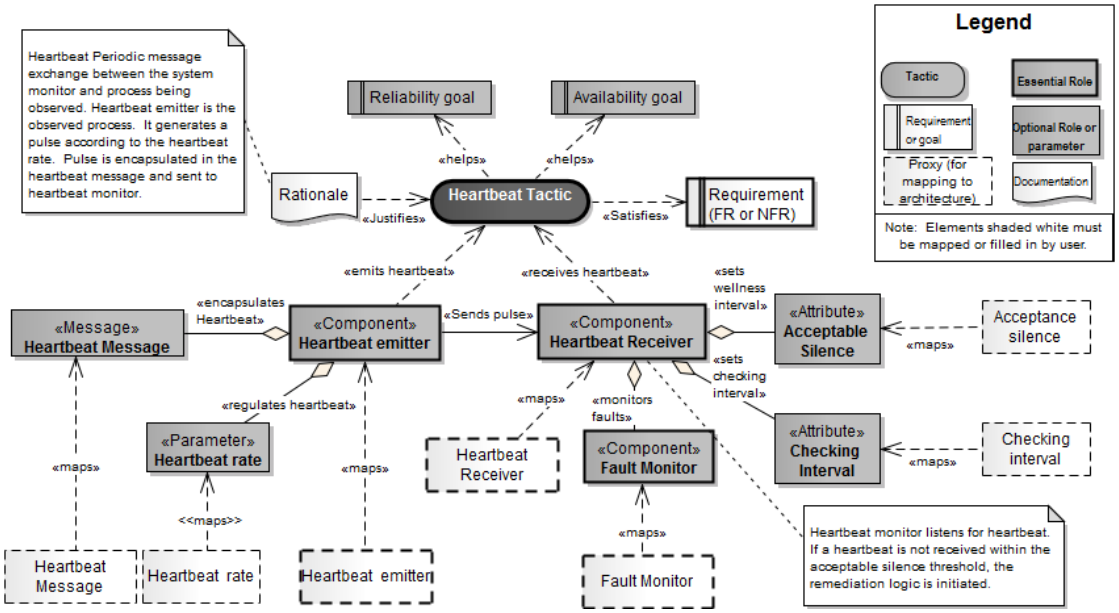


FIGURE 4.4: Traceability Pattern for Heartbeat Tactic

As an example, Figure 4.4 depicts the tTP developed for the *heartbeat* tactic. As previously explained Heartbeat is an architectural tactic used in many of dependable systems to monitor the availability of a critical process. This tactic contributes to the achievement of quality goals such as *reliability* and *availability*. The tTP associated with the *heart beat* tactic, defines three primary roles shaping this tactic as *receiver*, *emitter*, and *fault monitor*. In addition to these roles, this tTP identifies supporting parameters of *heart beat rate*, *heart beat message*, *checking interval*, and *acceptable silence threshold*. *Heart beat rate* indicates the frequency with which the heartbeat message is emitted, *heart beat message* represents the data structure required to transmit the heart beat, *checking interval* represents the frequency with which the receiver must check for the incoming heartbeat message, and finally the *acceptable silence threshold* indicates the maximum interval that can elapse between observed heartbeats signals before a failure decision is made. The Heartbeat tTP, models each of these roles and parameters, along with their relevant proxies connected via ≪maps≫ links. The proxies in tTP are used for creating traceability links to the concrete architecture.

The tTP conveys generic knowledge about architectural tactics through modeling structural relationships between roles and parameters. For example, the Heartbeat tTP depicts that, the *heart beat emitter* sends a pulse to the *heartbeat receiver*, and the *emitter* manages both the *heart beat rate* and the *heart beat message*.

Figures 4.5 through 4.7 depict examples of three additional tTPs representing *redundancy with voting*, *semantic based scheduling*, and *layers and partitions*. Each of these tTPs includes quality goals, rationales, requirements, and roles, as well as proxies for mapping the tTP to architectural elements.

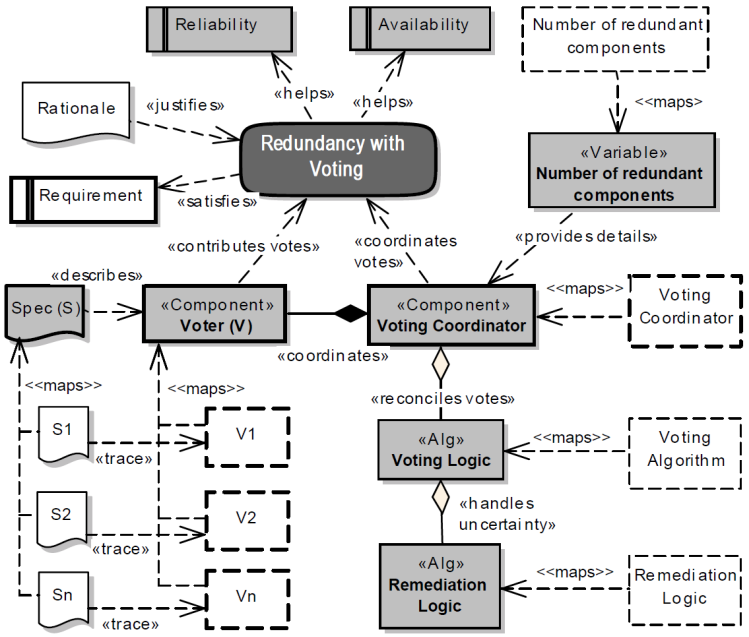


FIGURE 4.5: Tactic Traceability Pattern for Redundancy with Voting

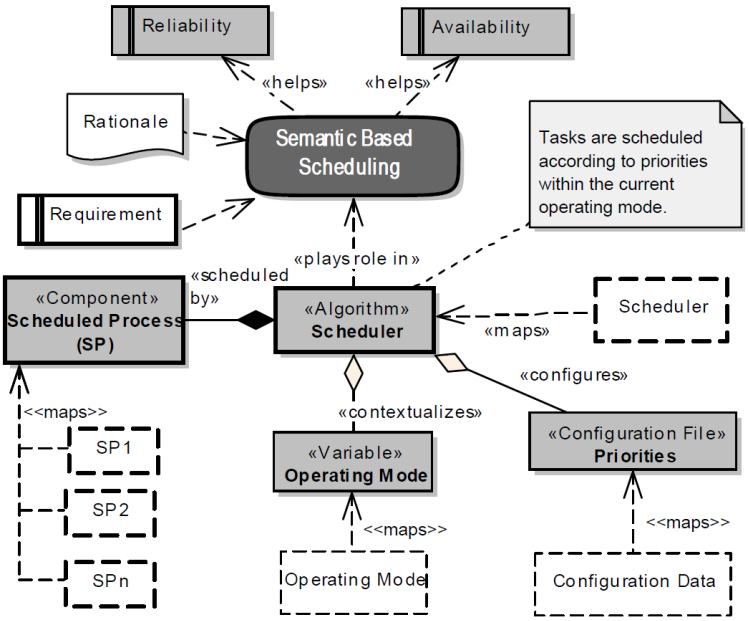


FIGURE 4.6: Tactic Traceability Pattern for Semantic Based Scheduling

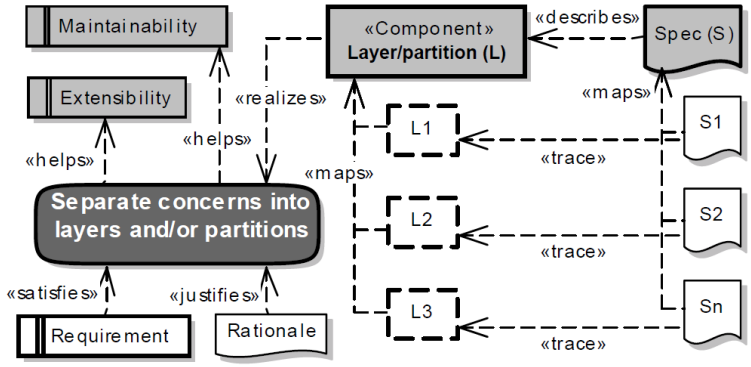


FIGURE 4.7: Tactic Traceability Pattern for Partitioning/Layers

One of the primary motivations behind designing a catalogue of tTPs is the support they provide for reducing the potential architectural erosion during the software maintenance process. This is done through two tasks of link creation and link usage which let us apply tTPs and benefit from them. During link creation, the architect/developer first identifies all the tactics used in the architecture of the system, he/she instantiates a tTP for each of these identified tactics, and then connects the tTP to the concrete implementation or design artifacts by mapping the proxies in the tTP to the relevant elements in the architecture and code.

Although the established trace links can be used to support manual inspections of the architecture, greater value can be realized through instrumenting an IDE to automate the use of the traceability links. Such automated tool should provide support for registering architectural elements mapped to tTP proxies, monitoring those elements for any changes during maintenance activities, and finally utilizing the infrastructure of the tTP to generate timely notifications to software maintainers to keep them informed of architectural decisions related to any components they are currently modifying.

An example of such a notification is provided in Figure 4.8. It illustrates a notification message that might be displayed if a developer modifies an architectural component mapped to the heartbeat tactic's emitter component. Visualizing tactics in this way abstracts out the important factors of the underlying design rationale, and makes design decisions understandable to software maintainers. A functioning prototype has been developed for this purpose. Using this prototype the developer can create a new tTP, establish trace links by mapping the proxies to the concrete elements in the source code. The established links are used by an event-based traceability [29] infrastructure to

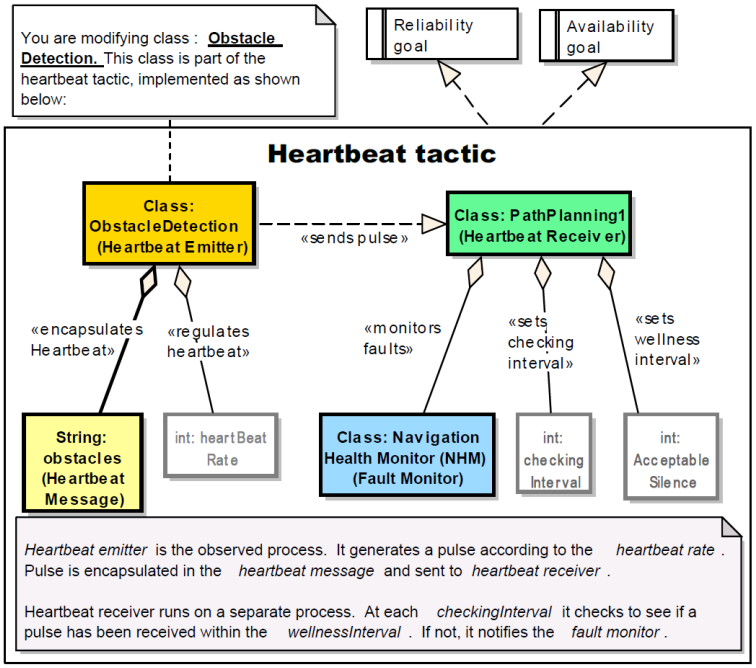


FIGURE 4.8: Design rationale displayed to user when they modify the heartbeat emitter component

monitor changes that a developer makes to source code. If a change impacts an element that traces back to a tactical role, the developer will be notified.

4.5 Examining the Research Questions

This section presents the results of two experiments conducted to investigate the use of tTPs in the software maintenance process. These experiment were designed to examine the following research questions:

- ***RQ1.*** *Does using tTPs potentially reduce the cost and effort of establishing and maintaining traceability links?*

- ***RQ2.*** *How useful are tTPs for notifying the developers of potential erosion through architecture-change impact analysis?*

The case study of a Lunar Robot system, reconstructed from an extensive set of publicly available NASA documents [70, 95] was conducted to examine the research questions. The detailed description of this case study, various design models and views, other artifacts such as list of requirements, and architectural decisions/tactics are described in details in Appendix A section A.2. The Lunar Robot system uses a reference architecture common to safety critical systems, therefore it provides a realistic environment for evaluating the utility of tTPs for preserving architectural qualities.

4.5.1 Examining RQ1. Reducing Cost and Effort

Counting the number of traceability links is a commonly adopted technique for estimating traceability effort. Therefore, we conduct an experiment to compare the number of traceability links needed to trace architectural tactics in the Lunar-Robot both with and without the use of tTPs. Consequently, two different traceability matrices were created. The first matrix, established traceability by using tTPs and mapping architectural elements to tTP proxies. The second matrix established traceability in a more traditional way without using tTPs. Similarly both traceability matrices provided forward and backwards traceability between architectural elements, tactics, rationales, goals, and requirements. In both of these matrices, the traceability links were created at a coarse-grained level to components, and also at a fine-grained level to the methods and/or parameters.

Eighteen different instances of tactics found in the Lunar Robot were considered to establish these two traceability matrices. The tactics included several cases of *heartbeat*, two cases of *redundancy*, three cases of *N Self-Checking Programming*, and several additional tactics as shown in Table A.2 of AppendixA.

Table 4.2 compares two traceability matrices generated in terms of the number of traceability links created for each tactic. The table shows the number of times each architectural tactic was adopted in the Lunar-Robot architecture, the number of links needed to support coarse grained traceability, both with and without the use of tTPs, and lastly the number of additional links needed to achieve fine-grained traceability to the method or parameter level using the tTP. As Table 4.2 depicts, Fine-grained links were only counted for scenarios which used the tTP. This is because such traces would be unlikely without the guidance of the tTP.

A pattern is dominant across all the results. In all cases, the use of tTPs reduced the number of traceability links which were required to establish and maintain in a specific project. On average, tracing each tactic through using a tTP required 5.4 links, while 9.28 were required for tracing tactics without tTPs. Moreover, these results showed that the total number of reusable links for a tactic plus the project-specific links can sometimes be greater than the number of links created without benefit of a tTP. This interesting observation highlights the fact that a catalogue of tTPs can be created once and reused across multiple projects. Also, the results show that the amount of additional effort needed to achieve fine-grained traceability is often very minimal given the existing infrastructure of each tTP. It should also be pointed out that the assumption is that equal effort is required to establish a traceability link using a tTP versus creating a link without the tTP. Furthermore, The additional links which are internal to the tTPs are not included in Ramesh's metamodel [104]. The unique benefits of the links is in providing a larger and richer set of semantically typed traceability links which could not be created without incorporation of the tTPs. Finally using tTPs replaces an ad-hoc typing of traceability links created by multiple stakeholders, with a simple mapping of an architectural element to a proxy in the tTP which provides a rich set of consistently typed traceability links.

All these results indicate that the answer to the research question of ***RQ1.*** *Does using tTPs potentially reduce the cost and effort of establishing and maintaining traceability links?* is positive. These observations have shown that the tTP simplifies the traceability task and therefore reduces the cost and effort of creating a traceability link. For two main reasons, we believe that the results of this case study is sufficient to draw a conclusion. Firstly, this is a real system not a toy example and the tactics covered in this example are realistic example of tactics which might be used in any industrial project. Secondly with out the loss of generality, a tTP implies the same benefit every time it is used. Therefore a single case study is enough to show usability of tTPs. A major reason for reduction of trace link is that tTPs change the task of link creation to mapping and therefore a set of trace links will be reused and not re-created. The links which are internal to a tTP are reused every time a tactic is traced and the trace user instead of creating all the links just maps the proxy to the design or code elements. In conclusion, based on the observed results in Table 4.2 using a catalogue of tTPs can lead to an effective traceability solution.

TABLE 4.2: Trace Link Counts per Tactic in the Lunar Robot

	Tactic	Number of occurrences of tactic	Coarse Grained Trace Links					Additional Fine-Grained Links	
			without tTP		with tTP			with tTP	
			per tactic	Total	per tactic	Total	re-used from tTP	Per Tactic	Total
1.	Heartbeat, Piggy backing and CRC check	6	9	54	4	24	6	4	24
2.	Redundancy with voting	1	7	7	3	3	6	6	6
3.	Active redundancy with degradation	1	17	17	9	9	6	5	5
4.	Multi Threading	1	12	12	9	9	4	1	1
5.	Separate processes with configuration files	1	19	19	9	9	5	2	2
6.	Layers (1)	1	9	9	7	7	4	2	2
7.	Layers (2)	1	11	11	9	9	Shared with Layers (1)	2	2
8.	Transaction	2	5	10	2	4	4	0	0
9.	N Self-Checking Programming w. Acceptance Tests	3	7	21	6	18	6	3	9
10.	N-Version Programming	1	7	7	6	6	4	3	3
		TOTALS:		167		98	45		

In summary, this experiment demonstrated that for our Lunar-Robot case study, the use of tTPs reduced the number of project-specific traceability links. Therefore the answer to the research question

RQ1. *Does using tTPs potentially reduce the cost and effort of establishing and maintaining traceability links?*

has been positive.

4.5.2 Evaluation RQ2. Usefulness of tTPs in Maintenance Scenarios

The second experiment was designed to examine research question of ***RQ2.***
This experiment tested whether tTPs provided adequate support for the goal of keeping developers informed of underlying architectural decisions during the maintenance process.

The software architecture change characterization framework defined by Williams and Carver has been used to develop different maintenance scenarios [131]. This framework addresses key areas involved in making changes to software architecture. It classifies the type of change as perfective, corrective, adaptive or preventative and identifies the characteristics of a software change that will have an impact on the high-level software architecture.

Table 4.3 shows eighteen change scenarios we have created using the above framework. Changes have had different scope and scale of impact. Some changes have impacted the whole architecture, such as in scenario #1 where a component is removed and existing ones are refactored. Some changes did not impact the architecture, instead they impacted the low level design, this is illustrated in Scenario #2 where the UI was modified. There are some changes that have not changed the architecture but resulted in changes in the source code, such as refactoring the code to eliminate the encryption method for messaging on the moon's surface (Scenario #6). Each one was simulated by creating change events, and then generating appropriate information displays. Table 4.3 also depicts the category of change covered by each maintenance scenario, the number of expected notification messages (as determined through a manual analysis), the number of successfully generated notifications (true positives), the number of unnecessary notifications (false positives), the number of missed notifications (false negatives), and the number of correctly ignored maintenance tasks (true negatives).

Among all 18 scenarios, nine of them affected tactical architectural decisions, and our prototype tool correctly recognized these potential impacts and generated appropriate notifications. The remaining nine scenarios indicated either micro-changes or functional changes that did not affect tactical architectural decisions. From all of these, our tool correctly filtered out three scenarios, but generated notifications for the remaining six. These six notifications represented false positives, meaning that tactic information was displayed, even though the actual change did not impact the tactic. Clearly, displaying unnecessary information can desensitize the developer to the importance

of such messages; however it should be noted that changes made to the large percentage of classes and components that are entirely unrelated to any tactic will never result in architectural notifications. False positives are therefore limited to classes which do contain or support an architectural tactic. Furthermore, false positives do not provide unrelated information, but do in fact provide information related to the general context of the change.

TABLE 4.3: Maintenance Scenarios

	Change Scenario			Motivation	Impact			Features
	NG	Imp	Description		Arch	Des	Code	
1.	Y	Y	New communication channel established directly between IVHM::Logger and the Robot Executive and with the CS::Task Sequencer & Dispatcher is removed	Refactoring, Performance Enhancement	*			Data transfer: Flow of data from system to external systems
2.	N	N	IVHM::Actuators Robust Regulators module communicates with CS::Robot Executive module instead of directly with the AVM.	Refactoring, Performance Enhancement	*			
3.	Y	N	IVHM::Simulation and Diagnosis Engine retrieves information from the shared repository instead of communicating with the Logger.	Refactoring: Performance Enhancement	*		*	
4.	Y	Y	Component Data Manager is moved from Navigation Domain process to Control System process and combined with component Mission Data Loader	Architecture refactoring	*	*	*	Data access: Receipt of data from external systems/repositories
5.	Y	Y	A security enhancement changes the encryption algorithm used to send mission information to agents on the LEO.	Secure data transmission		*	*	
6.	Y	N	Communication module is changed to eliminate encryption from messages sent between the Lunar Robot and the MCC whilst on the moon.	Performance enhancement			*	
7.	Y	Y	Rover's processor is upgraded to provide greater processing power. A mistake occurs and memory is accidentally reduced.	Enhancement	*			Devices: Hardware devices used by the system
8.	Y	Y	High resolution stereo cameras added to work with existing forward looking infrared cameras and laser sensors. The new data will be processed by modifying an existing algorithm in the obstacle detector component	Hardware enhancement	*	*	*	
9.	Y	Y	Multi-threading is used instead of multi-process for all the three domain of Navigation, Guidance and Control	Architecture refactoring	*	*	*	
10.	Y	N	Communication module is modified to remove a defect	Defect removal		*	*	System interface: Software interfaces with external systems
11.	N	N	Change in Operator Panel, modification of user interface, new menu item, changes look and feel of dialog	UI enhancement		*	*	human computer interaction interfaces
12.	Y	N	Change in protocol for communication with Mission Control Center (MCC) at the earth	Adaptive enhancement	*		*	Communication: Interfaces to other systems/data
13.	Y	Y	Mission data and operations are logged following completion of a task instead of following each simple action	Performance Enhancement			*	Computation:algorithm functions and modification of data
14.	N	N	Component Relative Navigation is modified to fix bugs	Defect removal			*	
15.	Y	Y	Change in functionality in a performance critical module (PPOD)	Addition of new functionality	*	*	*	
16.	Y	Y	Obstacle Detection component is merged into both Path Planning threads	Corrective maintenance/self checking	*	*	*	
17.	Y	N	Data stored by the IVHM::Logger in the Robot's blackbox, is modified to be stored in encrypted format	Security			*	I/O format of information processed by system
18.	Y	N	IVHM Simulation and Diagnosis Engine is modified to decode data stored in encrypted format in the blackbox	Security	*	*	*	

*'NG' = Notification generated due to potential impact of the change; 'Imp' = a true risk of affecting an underlying architectural decision. Hence (Y,Y)= True positives, (N,N) = True negatives, (N,Y)=False negatives, and (Y,N) = False positives.

4.6 Summary

This chapter first reports the results of an extensive study of architectural decisions conducted in highly dependable and complex avionic systems. The conducted study involved reviewing the specifications of several high-assurance software systems including the Airbus A320/330/340 family, Boeing 777, Boeing 7J7 [11, 113] , NASA robots [70, 95], NASA Crew Exploration Vehicles [65, 115, 116] and also the implemented code of performance-centric systems such as Google Chromium OS, Hadoop Framework etc. As a result we summarized some of the important issues facing architectural traceability and presented a meta-model for tracing architecturally significant requirements. To our knowledge, these issues have not previously been addressed in such a systematic manner, and as a result existing traceability approaches for tracing architectural concerns and supporting design rationales tend to suffer from well documented maintenance, and usage problems.

In this chapter, we proposed the generic decision-centric meta-model to trace quality concerns into design and implementation artifacts. This meta-model suggests tracing the requirements through design decisions into design components and implementation modules. We extend the notion of the meta-model for a representative selection of architectural tactic, and proposed an augmented model called tTP in which, it is clear where to create traceability links in order balance the costs versus benefits of tracing architectural concerns. As our results have indicated, tTPs reduce the cost and effort of traceability through providing a set of re-usable traceability links. However, upfront effort is required to create the tTPs. Clearly our approach is constrained by the extent to which similar tactics are reused across projects.

"A computer would deserve to be called intelligent if it could deceive a human into believing that it was human."

Alan Turing

Chapter 5

Automated Trace Generation

Manually tracing architectural decisions into the code, can be prohibitively expensive as it may involve creating and maintaining an almost impossible number of traceability links; but on the other hand, failing to trace architectural concerns leaves the system vulnerable to problems such as architectural degradation.

Contribution: A cost effective approach which automates construction of traceability links for architectural tactics can be significantly beneficial. To achieve this aim, we present a novel approach which utilizes machine learning methods and lightweight structural analysis to detect tactic-related classes. The detected tactic-related classes are then mapped to a tactic Traceability Pattern (tTP) by achieving traceability in in an automated way. We train our trace classifier using code extracted from fifty performance-centric and safety-critical open source software systems.

To achieve the *Automate Trace Generation* goal we first, introduce and examine set of algorithms and processes designed to automatically reconstruct traceability links for architectural tactics. This builds on the tTPs proposed in the previous chapter which required all traces to be created manually. Second, we evaluate the effectiveness and applicability of our classifier for an industrial setting through an extended study of a large scale software system.

5.1 Proposed Approach

The problem of detecting architectural tactics in source code is an increasingly challenging task. This is mainly because of the nature of architectural tactics and the ways and forms in which they are implemented. In the literature, there have been various techniques for detecting *design patterns* in the source code, unfortunately these approaches can not be applied to detect architectural tactics as design patterns are different from tactics in scope of impact, level of abstraction and the type of concerns they address. Therefore detection of tactics turns out to be a more challenging task. More specifically, Design patterns tend to be described in terms of classes and the associations among them [51], while it is unlikely for tactics to be described in terms of roles and interactions [81]. This means that a single tactic could potentially be implemented using a variety of different design patterns or proprietary designs. For instance, our studies of the heartbeat tactic in real systems showed that this tactic has been implemented using (i) direct communication between the emitter and receiver roles *(found in Chat3 and Smartfrog systems)*, (ii) the observer pattern [51] in which the receiver registered as a listener to the emitter *found in the Amalgam system*, (iii) the decorator pattern [51] in which the heartbeat functionality was added as a wrapper to a core service *(found in Rossume and jworkosgi systems)*, and finally (iv) numerous proprietary formats that did not follow any specific design pattern.

Architectural tactics are not dependent upon a specific structure, or structural constraints similar to those found in design patterns, therefore, we cannot use structural analysis as the primary means of identification. To detect the tactics in the code in an automatic way, we propose an approach which relies primarily on information retrieval (IR) and machine learning techniques to train a classifier to recognize specific terms that occur commonly across implemented tactics, however we also use light-weight structural analysis to support the differentiation of specific tactic roles.

Figure 5.1 depicts the steps involved in the proposed process. First, a tactic-classifier identifies all classes related to a given tactic, and then creates tactic-level traceability through mapping those classes to the relevant tactic. Second, we use a more finely-tuned classifier which is enhanced by a lightweight structural analysis package to identify the roles of each tactic as defined in tTP. For example, in the case of the *heartbeat* tactic, the classifier attempts to identify *heartbeat emitter* and *heartbeat receiver* roles, or in the case of the *voting* tactic, it attempts to identify *voting coordinators*

and *voters*. The detected tactic-related classes are then mapped to tactic Traceability Patterns (tTPs) which connect the classified classes back to the design rationales, requirements, and other related artifacts.

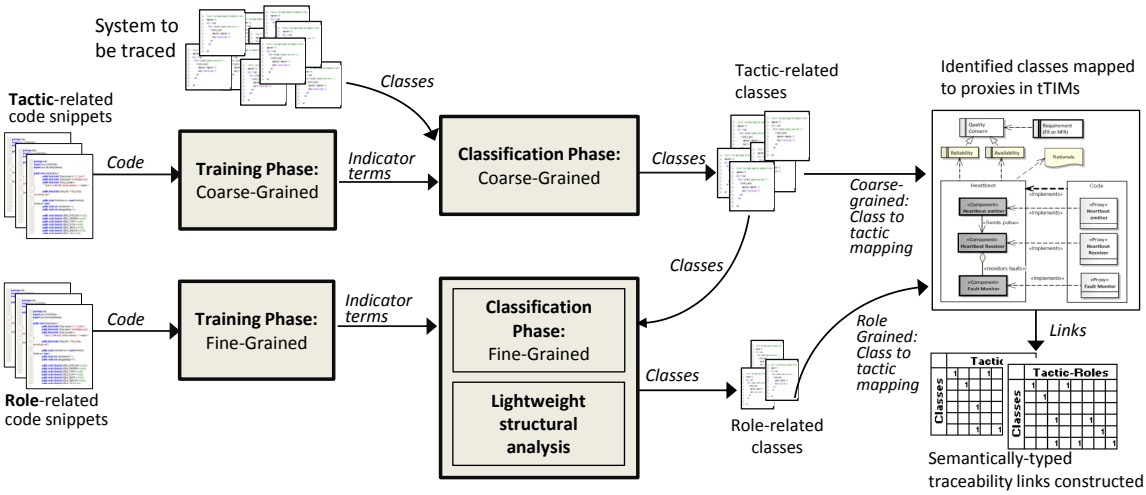


FIGURE 5.1: An Overview of the Tactic-Related Trace Reconstruction Process

The classification of classes into various tactics has been achieved by utilizing a text classification technique previously used in other domains such as classification of textual non-functional requirements and for tracing them to software regulations. [37] [32].

Typically classifiers include three phases of preparation, training, and classifying which are defined as follows:

Phase i. Preparation. Standard information retrieval techniques are used to pre-process all the data and transform each class into a vector of terms.

Phase ii. Training. In the training phase, the classifier learns the key terms commonly used to implement each architectural tactic. In this phase, the classifier takes a set of pre-classified code segments as input, and produces a set of indicator terms that are considered representative of each tactic type. For instance, the term of *priority*, is found more commonly in code related to the *scheduling* tactic than in other kinds of code, and therefore the classifier gives it a higher weighting with respect to scheduling tactic.

More formally, let q be a specific tactic such as *heart beat*. Indicator terms of type q are mined by considering the set S_q of all classes that are related to tactic q. The cardinality of S_q is defined

as N_q. Each term t is assigned a weight score $Pr_q(t)$ that corresponds to the probability that a particular term t identifies a class associated with tactic q. The frequency $freq(c_q, t)$ of term t in a class description c related with tactic q, is computed for each tactic description in S_q. $Pr_q(t)$ is then computed as:

$$Pr_q(t) = \frac{1}{N_q} \sum_{c_q \in S_q} \frac{freq(c_q, t)}{|c_q|} * \frac{N_q(t)}{N(t)} * \frac{NP_q(t)}{NP_q} \tag{5.1}$$

Phase iii. Classification. During the classification phase, the indicator terms computed in Equation 5.1 are used to evaluate the likelihood ($Pr_q(c)$) that a given class c is associated with the tactic q. Let I_q be the set of indicator terms for tactic q identified during the training phase. The classification score that class c is associated with tactic q is then defined as follows:

$$Pr_q(c) = \frac{\sum_{t \in c \cap I_q} Pr_q(t)}{\sum_{t \in I_q} Pr_q(t)} \tag{5.2}$$

where the numerator is computed as the sum of the term weights of all type q indicator terms that are contained in c, and the denominator is the sum of the term weights for all type q indicator terms. The probabilistic classifier for a given type q will assign a higher score $Pr_q(c)$ to class c that contains several strong indicator terms for q. Classes are considered to be related to a given tactic q if the classification score is higher than a selected threshold.

We applied the classifier in different ways to detect architectural tactics at multiple level of granularity, to examine the following research questions:

- ***RQ3.*** *How accurately does the Tactic Detector generate trace links using two different training methods of tactic descriptions and code snippets? Which one produces better classification results?*
- ***RQ4.*** *How effectively can the Tactic Detector identify tactic-related classes for the five targeted tactics in HADOOP?*
- ***RQ5.*** *How accurately does the Tactic Detector generate role-level trace links for each architectural tactic?*

The following sections present the design and evaluation results of these experiments. Each experiment design involved creation of a specific data set which were created over the course of 7 months. The results of experiments and examination of research questions were evaluated using four standard metrics of recall, precision, F-Measure, and specificity computed as follows where *code* is short-hand for *code snippets*.

$$Recall = \frac{|RelevantCode \cap RetrievedCode|}{|RelevantCode|} \tag{5.3}$$

while precision measures the fraction of retrieved code snippets that are relevant and is computed as:

$$Precision = \frac{|RelevantCode \cap RetrievedCode|}{|RetrievedCode|} \tag{5.4}$$

Because it is not feasible to achieve identical recall values across all runs of the algorithm the F-Measure computes the harmonic mean of recall and precision and can be used to compare results across experiments:

$$F - Measure = \frac{2 * Precision * Recall}{Precision + Recall} \tag{5.5}$$

Finally, specificity measures the fraction of unrelated and unclassified code snippets. It is computed as:

$$Specificity = \frac{|NonRelevantCode|}{|TrueNegatives| + |FalsePositives|} \tag{5.6}$$

5.2 Tactic Level Link Reconstruction

The initial step for reconstructing tactic-related traceability links applied the classifier described in Equations 5.1 and 5.2 to recognize and detect classes which implement the desired tactic. Two different experiments were conducted to investigate different training methods. In the first experiment, the training method involved using *tactic descriptions* to train the classifier, while the second experiment used actual *code snippets* taken from classes implementing each of the tactics for the

training purpose. Both these experiments were repeated using a variety of term thresholds and classification thresholds. Due to the significant cost and effort of manually constructing the 'answer sets' needed to evaluate our approach against non-trivially sized projects, we limited the work described here to the *heartbeat*, *scheduling*, *resource pooling*, *authentication*, and *audit trail*. These tactics were selected because they represented a variety of reliability, performance, and security concerns.

The main objective behind designing and conducting these two experiments were to examine whether the classification method described in Equations 5.1 and 5.2 could be used to identify tactic-related classes for the five targeted tactics, and also to determine whether the tactic descriptions or the code snippets produced better classification results. We hypothesized that the code-trained classifier would be more effective for retrieving tactic-related classes.

5.2.1 Experiment 1: Training with tactic descriptions

In this experiment, for each of the five targeted tactics i.e. *heartbeat*, *resource pooling*, *scheduling*, *audit trail*, and *authentication*, we established a dataset containing ten descriptions of tactic taken from text books, online descriptions, and publications. For training purposes, the dataset also included 20 descriptions of non-tactic-related IT documents so that the common IT terms could be filtered out from trained indicator terms. The following text provides an excerpt from a description for the *audit trail* tactic:

> *A record showing who has accessed a computer system and what operations he or she has performed during a given period of time. Audit trails are useful both for maintaining security and for recovering lost transactions.....*

In order to build the *testing set*, for each of the five tactics we identified 10 different open-source projects, which incorporated the tactic in their design and implementation. From each of these projects the code segments that were implementing the tactic were retrieved. Furthermore, four additional non tactic-related classes were retrieved for testing purposes.The following code represents two methods extracted from a code snippet for the audit tactic.

TABLE 5.1: Indicator terms learned during training method 1

Tactic Name	**Document trained indicator terms**
Heartbeat	heartbeat, fault, detect, messag, period, watchdog, send, tactic, failur, aliv
Scheduling	prioriti, schedul, assign, process, time, queue, robin higher, weight, dispatch
Authentication	authent, password, kerbero, sasl, ident, biometr, verifi, prove, ticket, purport
Resource Pooling	thread, pool, number, worker, task, queue, executor, creat, overhead, min
Audit Trail	audit, trail, record, activ, log, databas, access, action, monitor, user

```
setAuditUserIdentity(inUi);
setAuditSequenceNumber(inSeqNum);
setAuditMachineOfOrigin(inMachineOfOrig);
setAuditDateTime(inDateTime);

public boolean isAuditUserIdentifyPresent(){
   return(this.auditUserIdentify != null);
}
public BigDecimal getAuditSequenceNumber(){
   return this.auditSequenceNumber;
}
```

In this experiment, we first trained the classifier using the descriptions of tactics and then tested the trained classifier against the testing set (extracted code snippets). The experiment was repeated using a variety of term thresholds and classification thresholds.

Table 5.1 shows the top ten indicator terms that were learned for each of the five tactics using tactic descriptions training techniques.

The results of running the description trained classifier over tactics' code snippets is shown in Figure 5.2. We reported the F-Measure results for several combinations of term and classification threshold values. In four of the five cases, namely *scheduling*, *heartbeat*, *audit*, and *pooling* the F-Measure was higher than 0.70 which could be interpreted as an acceptable result. One phenomenon that needs explaining in these graphs are the horizontal lines in which there is no variation in F-Measure score across various classification values. This generally occurs when all the terms scoring over the term threshold value also score over the classification threshold.

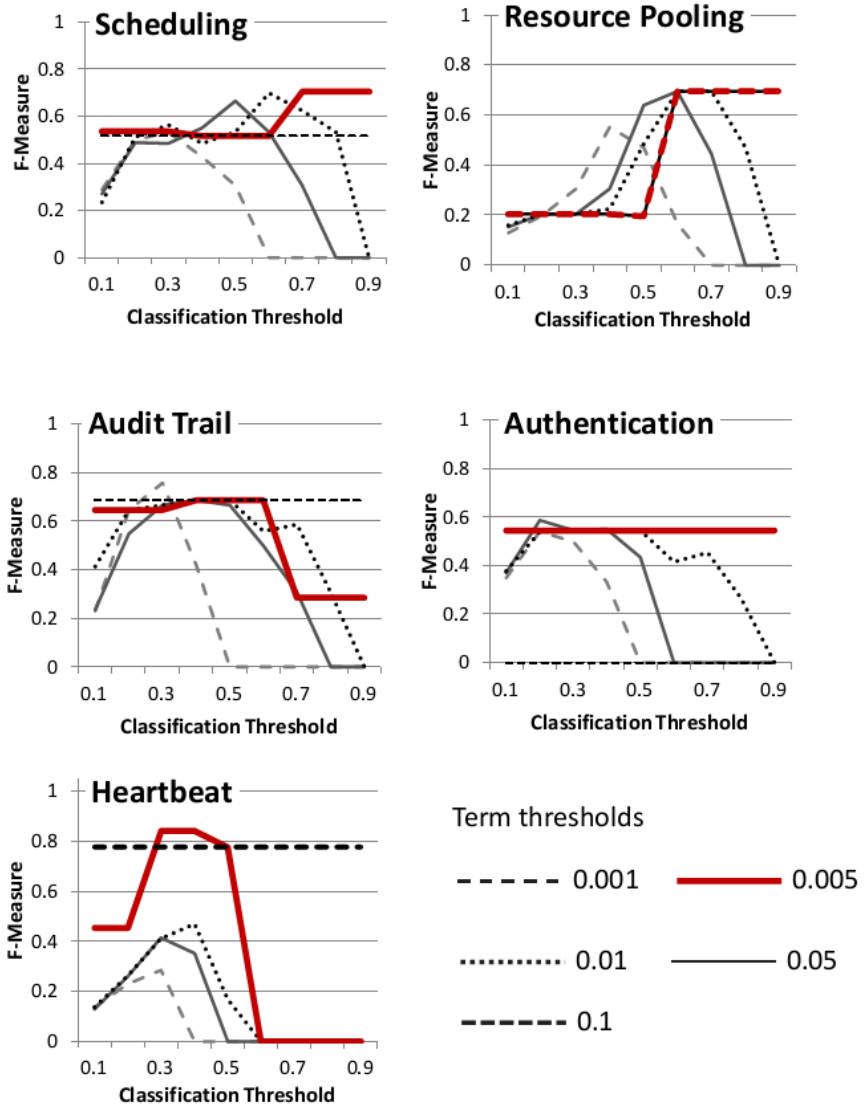


FIGURE 5.2: Experiment 1: Trained using Tactic Descriptions

5.2.2 Experiment 2: Training with code snippets

The second experiment involved training the classifier using code snippets. We used the code based dataset already created in experiment(1). Basically for each of the five targeted tactics, 10 different open-source projects, in which the tactic was implemented were identified and code segments that were closely related to the tactic were extracted. We also retrieved four additional non tactic-related classes for training and testing purposes.

TABLE 5.2: Indicator terms learned during training

Tactic Name	**Code trained indicator terms**
Heartbeat	heartbeat, ping, beat, heart, hb, outbound, puls, hsr, period, isonlin
Scheduling	schedul, task, prioriti, prcb, sched, thread, , rtp, weight, tsi
Authentication	authent, credenti, challeng, kerbero, auth, login, otp, cred, share, sasl
Resource Pooling	pool, thread, connect, sparrow, nbp, processor, worker, timewait, jdbc, ti
Audit Trail	audit, trail, wizard, pwriter, lthread, log, string, categori, pstmt, pmr

Table 5.2 shows the top ten indicator terms that were learned for each of the five tactics using the code snippets training technique. Although there is significant overlap, the code-snippet approach unsurprisingly learned more code-oriented terms such as *ping*, *isonlin*, and *pwriter*.

Because of the time-consuming nature of finding and retrieving architectural tactics from real systems, we adopted a standard 10-fold cross-validation process in which the code-snippets dataset served as both the training and testing set. This dataset has 10 projects, where from each project we have 1 code snippet implementing a tactic and four unrelated code-snippets. Typically 10-fold cross-validation experiment has 10 executions which in each, the data was partitioned by project such that in the first run nine projects were used as the training set and one project was used for testing purposes. Following ten such executions, each of the projects was classified one time. The experiment was repeated using the same pairs of term thresholds and classification thresholds used in the previous experiment.

Figure 5.3 reports the F-Measure results for the 10-fold cross validation technique over the code trained classifier using several combinations of threshold value. In three of five cases *scheduling*, *authentication*, and *pooling* the F-Measure was higher than 0.85. Also the other two remaining tactics were close to 0.80. Comparing these results with previous experiments we observe that in four of the five cases, *scheduling*, *authentication*, *audit*, and *pooling* the code-trained classifier outperformed the description-trained classifier. In the case of *heartbeat*, the description-trained classifier performed better at term threshold values of 0.05 and classification thresholds of 0.3 to 0.4.

Experiment 3. Industrial Case Study

The third experiment aimed to reconstruct tactic-related traceability links in a real large scale software system. Therefore the Apache Hadoop software framework, a system which supports distributed processing of large datasets across thousands of computer clusters was selected as a case study. The Hadoop library includes over 1,700 classes and provides functionality to detect and

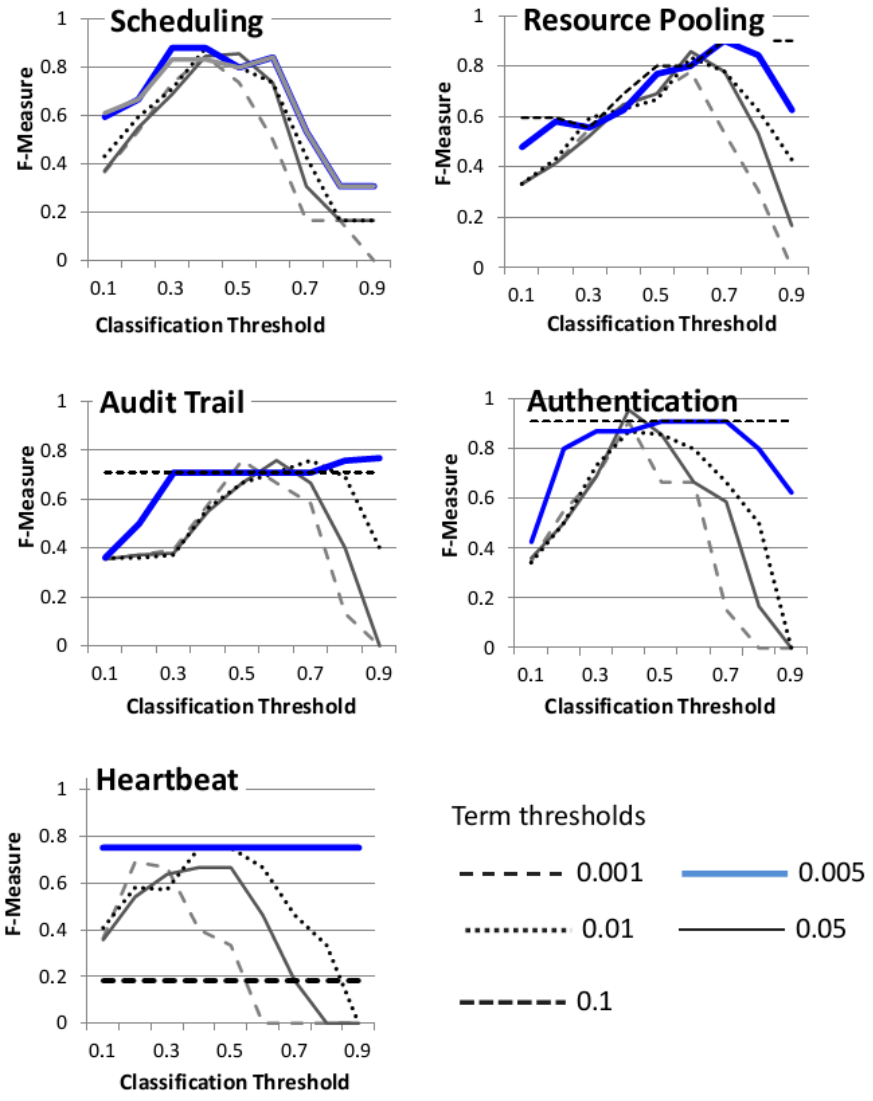


FIGURE 5.3: Experiment 2: Trained using Code Snippets from Tactics implemented in Open Source Systems

handle failures in order to deliver high availability service even in the event that underlying clusters fail.

The detailed information about Hadoop's Architecture and tactics adopted in this system can be found in Appendix A.3. In order to conduct the experiment(3), the first step included building an 'answer set' for evaluation purposes by manually identifying *heartbeat*, *resource pooling*, *scheduling*, *audit trail*, and *authentication* tactics in Hadoop. Creation of the answer set was accomplished by (i) reviewing the available Hadoop literature[1] to look for any references to specific tactics, and then

manually hunting for the occurrences of those tactics in the source code, (ii) browsing through the Hadoop classes to identify tactic-related ones, (iii) using Koders (search engine) to search through the code using key terms (to reduce bias, this search was performed by two researchers in our group prior to viewing the indicator terms generated during the classification training step), and finally (iv) posting a question on the Hadoop discussion forum describing the occurrences of tactics we found, eliciting feedback, and acquiring confirmation from Hadoop developers. The results of these activities are documented in the Appendices A.3. Table A.3 shows the occurrences of the five tactics we identified in Hadoop, and which were then used as the 'answer set' for the remainder of the case study. Our analysis showed that 1,557 classes were not tactic related, 145 classes implemented one tactic only, 14 classes implemented two tactics, two classes implemented one tactic, and one class implemented four tactics.

The results observed in Experiment(1) and Experiment(2) were used to decide the optimum thresholds to execute the case study in a way that promised the best outcome. Based on these results, in order to achieve high recall levels in the case study, we decided to use the code-trained classifier developed in our previous experiments with the following threshold levels (i.e. term threshold of 0.001 and classification threshold of 0.5) to classify all 1,700 classes in Hadoop according to the five targeted tactics.

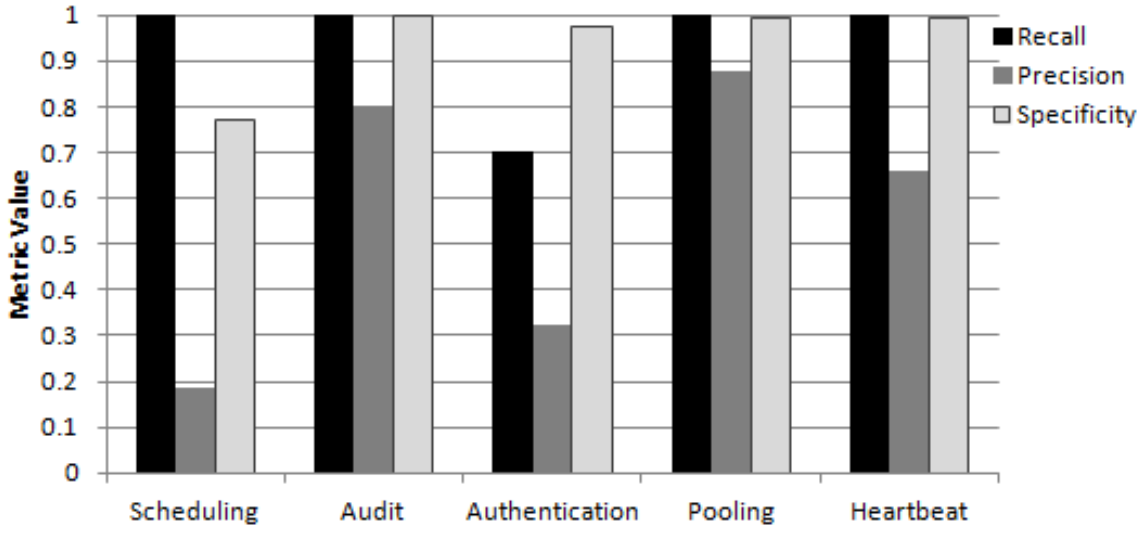


FIGURE 5.4: Results for Coarse-Grained Tactic Traceability in Hadoop

The above thresholds were used to classify all the 1,700 Java files in Hadoop Framework. The results, reported in terms of Recall, Precision, Specificity, and F-Measure are depicted in Figure 5.4. These

results indicate that we were able to correctly reject approximately 97-99% of the unrelated code snippets in each of the cases. In three cases of *audit*, *resource pooling*, and *heartbeat* we were able to recall all of the related code snippets; however for *scheduling* and *authentication* we were only able to recall 87% and 70% of the related code snippets respectively In all five cases precision ranged from 60% to 87%.

5.3 Role Level Link Reconstruction

In the previous section we examined the performance of a classifier for establishing tactic level traceability through different training techniques. This section describes the approach we took to train a classifier to differentiate between various tactic roles defined by a tTP. To build such a classifier we constructed a *Role Snippets Dataset*. This dataset was a new *code snippets* dataset in which we collected several separate code snippets for each of the tactic's roles. For example, each project for the *scheduling* tactic included one code segment implementing the *scheduler* role, one code segment implementing the *scheduled by* role, as well as four unrelated code segments.

The previously described 10-fold cross-validation experiment was repeated with the role-based training methods to see if we could effectively retrieve classes according to their role in the project. Initial experiments and observations indicated that the terms used across roles in a given tactic were quite similar and so differentiation was poor. To tackle the observed problems an extensive exploratory investigation was conducted to determine the best possible solution to classify Java classes by roles; Here we report only the final technique that was adopted.

5.3.1 Light Weight Structural Analysis

The classifier described previously is used in the first two steps of this process, while steps three to five utilize a light-weight structural analysis developed through extensive analysis of the tactic-related code found in Fault Tolerant CORBA, the Google Chromium OS and the ROSSUME robotic system. As a result of this analysis, we hypothesized that utilizing class hierarchy information and class dependencies caused by method calls and method invocations could further improve the quality of tactic traceability. Therefore we proposed a hybrid approach for detecting tactic's roles. The resulting technique is specified as follows:

1. The tactic-grained classifier is first run against the entire set of classes in order to identify an initial set of tactic related classes for each tactic.

2. The role-grained classifier is then run against the subset of classes returned by the tactic-grained classifier. Following this step, each of these classes is assigned a probability with respect to each of the tactic related roles.

3. Based on observations that tactic related-behavior is often specified in base classes, probabilities are propagated across "extends" relationships if the probability in the base class for a specific tactic role is higher than that of the derived class. Values are not propagated across "implements" relationships because classes that implement an interface define their own behavior.

4. Based on observations that most tactics require communication between roles, dependency analysis is performed to eliminate classes that do not interact with other tactic classified classes. For example, a class assigned some probability of being a heartbeat receiver is in fact unlikely to actually play that role unless it is associated with other classes which are also classified as heartbeat-related. However, this heuristic is not valid for all tactics, as some tactics might implement roles using inbuilt class libraries. For example resource pooling might be implemented using the classes from Java.util.concurrent, meaning that it is possible to have a tactic-related, yet isolated class. Furthermore, in the case that standard library functions are used in this way, it becomes relatively trivial to identify the occurrence of such a tactic. For purposes of our study, we therefore apply this heuristic to all tactics apart from resource pooling.

5. Wherever feasible, classes are placed into functional groupings according to their associations, so that different instances of the same tactic can be separated out.

6. Finally, classes are classified according to the role with the highest probability score, as long as that score is higher than a predetermined threshold.

Beside these 6 steps, we also explored other options for structural analysis, such as method signature, direction of method calls, indirect dependencies, and many other types of dependencies, For example, while it might seem reasonable to differentiate between a heartbeat *sender* and *receiver* according to the direction of the heartbeat message, the variety of implementations made this quite difficult.

In order to evaluate the light-weight structural approach, we used Hadoop case study. The role based code snippets needed in experiments could not be used for the purpose of evaluation as they did not carry associated structural information. We therefore conducted an evaluation within the richer context of the Hadoop Framework.

5.3.2 Role-Level Trace Reconstruction in a Real Case Study

The role-grained classifier was used to classify the tactic-related classes by role. Based on initial analysis of results and gained intuition, the classification threshold (i.e. the minimum threshold needed to classify a class according to a specific type) was set at 0.5. Figure 5.5 reports these results and indicates that in each case, the fine-grained classifier was able to classify one dominant role better than the other one. For instance, in the *scheduling* tactic the *"scheduler"* role tended to contain more tactic-specific terms than the *"scheduled by"* role, and was therefore classified more accurately.

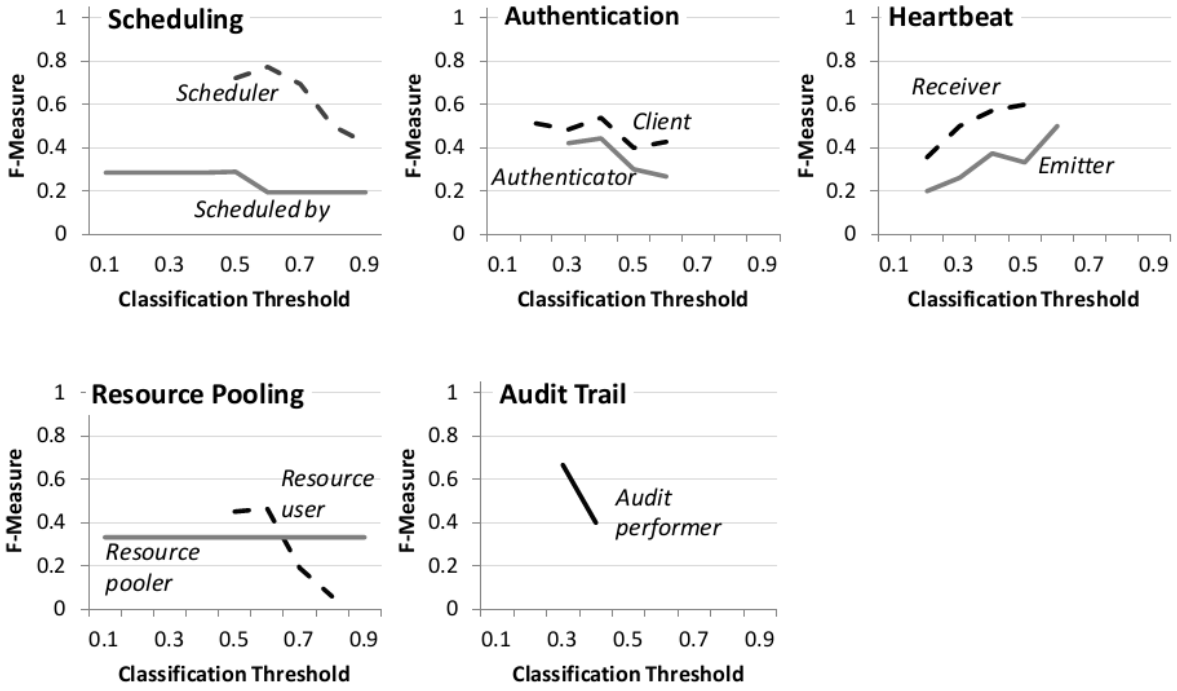


FIGURE 5.5: Results for Fine-Grained Tactic Traceability in Hadoop

A more detailed and specific explanation of these results is shown through an example of one of the heartbeat instances in Hadoop. Figure 5.6 depicts the role-based classification for the heartbeat tactic used in Hadoop's HDFS subsystem.

Tactic roles are depicted as ≪emitter≫ or ≪receiver≫ stereotypes and are also shaded in gray. For example DataNode which implements DatanodeProtocol sends the heartbeat message to the NameNode, therefore each of them has the ≪emitter≫ stereotype. In Figure 5.6, roles are ordered according to probability for each class, and if all probabilities fall below the classification threshold

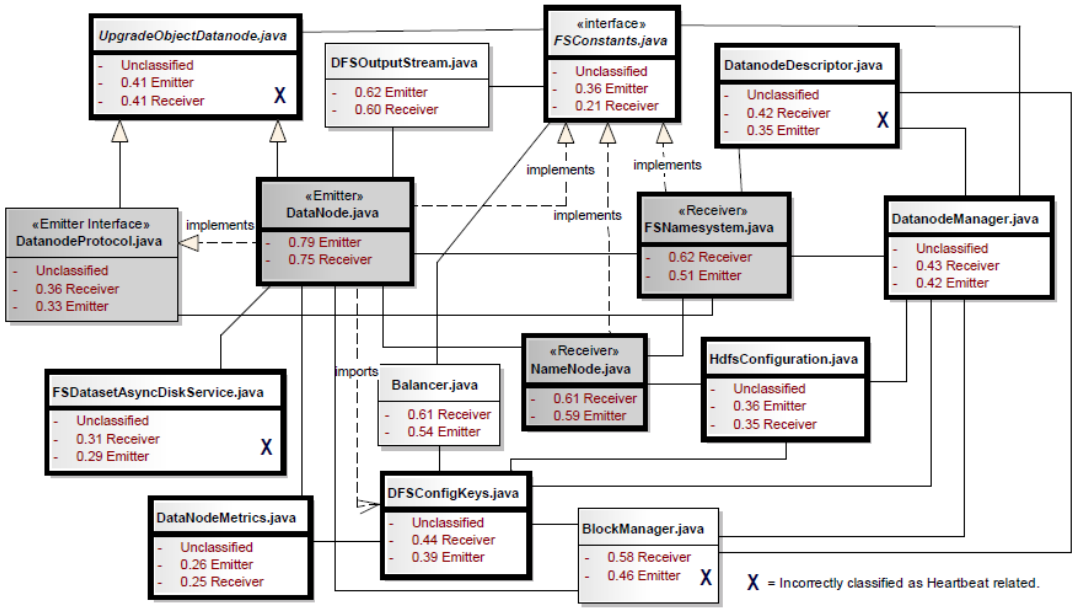


FIGURE 5.6: Reverse Engineered Role-Grained Traces for a Heartbeat Tactic in Hadoop

an additional *unclassified* role is added. All classes with bold borders have been correctly classified either as a specific tactic role or as unclassified. As depicted in the diagram, we were able to correctly classify two out of three receivers, one out of two emitters, and to correctly reject eight out of 11 unclassified classes. The missed emitter was in fact an interface and not a fully defined class. Classes originally misclassified by the tactic-grained classifier as heartbeat related are marked with an **X**.

As described early in this chapter, the last part of Role Level link construction is mapping the detected roles to their corresponding proxies in the tTP. This step will provide fine-grained traceability links with enhanced semantics. The link semantic is achieved through the information embodied in each tTP. As it is depicted in Figure 5.7, a subset of role-classified classes are mapped to specific roles in a tTP, and other classes which were classified as tactical but unclassified to any role, are mapped at the tactic level. These mappings are performed automatically as part of the classification process, and as a result, the classified classes are traced to other tactic-related classes, to quality goals, and to related requirements. For example, in this case the mapping of *DataNode.java* as a *Heartbeat emitter* and *FSNamesystem.java* as a *Heartbeat receiver* establishes a relationship between them of type *Sends Pulse*. Similarly it establishes that both java classes contribute to achieving the reliability requirement that "HDFS must store reliability even in the presence of failures."

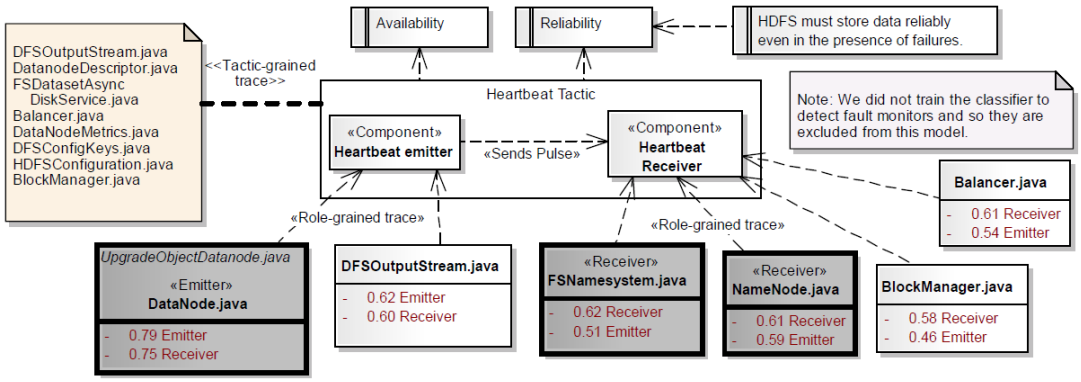


FIGURE 5.7: Trace Reconstruction through Mapping Classified Classes at both Tactic and Role Granularities to a tactic Traceability Pattern

5.4 Examining the Research Questions

This section summarizes the results of experiments conducted in the previous sections and answers whether the research questions related to the goal of automating trace generation. In total, we conducted four different experiments to examine effectiveness of the proposed solutions for automating tTP based traceability. These experiment were designed to examine the following research questions as defined in chapter 1:

- ***RQ3.*** *How accurately does the Tactic Detector generate trace links using two different training methods of tactic descriptions and code snippets? Which one produces better classification results?*
- ***RQ4.*** *How effectively can the Tactic Detector identify tactic-related classes for the five targeted tactics in HADOOP?*
- ***RQ5.*** *How accurately does the Tactic Detector generate role-level trace links for each architectural tactic?*

In order to answer research question ***RQ3*** we created a table (Table 5.3) which reports the results from the use of the two different training methods of *Tactic-Descriptions* and *Code-Snippets* to train the tactic level classifier. These experiments were run for various term and classification thresholds. In order to compare the results of these two experiments, we report only a result which achieved the high levels of recall (0.9 or higher if feasible) while also returning as high precision as possible.

TABLE 5.3: A Summary of the Highest Scoring Results

Tactic	Training Method	F-Measure	Recall	Prec.	Spec.	Term/ Classification threshold
Audit	Descript.	0.758	1	0.611	0.972	0.001 / 0.3
	Code	0.758	1	0.611	0.833	0.001 / 0.5
Authentication	Descript.	0.588	1	0.416	0.945	0.005 / 0.2
	Code	0.956	1	0.916	0.977	0.005 / 0.4
Heartbeat	Descript.	0.75	0.6	1	1	0.01 / 0.4
	Code	0.689	1	0.526	0.775	0.001 / 0.2
Pooling	Descript.	0.695	0.8	0.615	0.98	0.005 / 0.6
	Code	0.9	0.818	1	1	0.05 / 0.7
Scheduling	Descript.	0.705	0.545	1	1	0.05 / 0.8
	Code	0.88	1	0.785	0.931	0.01 / 0.4

These results show that in four cases the code-trained classifier recalled all of the tactic related classes, while also achieving reasonable precision for most of the tactics(.0.526 for Heartbeat to 0.916 for Authentication). The description-trained classifier achieved recall of 1 for only two of the tactics and its precision was changing from 0.416 for Authentication to 1 for Scheduling and Heartbeat. In all cases except Heartbeat tactic, the code based training method achieved a higher level for F-Measure.

The results showed that the code snippets trained classifier generally outperformed the description trained classifier. This can be concluded from the higher F-Measure which we observed in four out of five cases trained using the code snippets.

The results of the experiment conducted on the Hadoop case study to reconstruct tactic level trace links indicates that the proposed approach performed reasonably well on a real case study. These results showed that for four of the five tactics we were able to recall 100% of the tactic-related classes at precision values ranging from 18-88%. In the case of the authentication tactic we were only able to recall 60% of the tactic-related classes. Considering the search space for architectural tactics in a large project such as Hadoop, the answer to research question ***RQ4*** can be interpreted as positive. While at the same time, the last experiment on using hybrid approach to reconstruct role-level trace links did not achieve a high level of F-Measure for all the roles in a tactic, and suffered from a large number of false positive cases. Therefore we are not able to have a positive answer for ***RQ5*** and conclude that constructing role-level trace links using our approach was successful.

5.5 Summary

This chapter has presented a technique for automating the reconstruction of traceability links between classes and their related architectural tactics. It has been demonstrated and evaluated within the context of a large-scale performance-centric software system. Integrating the concept of tTPs with existing notions of trace retrieval and classification introduces a novel approach to tracing architectural concerns, and minimizes the human effort required to establish traceability. It produces traces which can be used to support critical software engineering tasks such as software maintenance and ultimately to help mitigate the pervasive problem of architectural erosion.

"I think a hero is an ordinary individual who finds strength to persevere and endure in spite of overwhelming obstacles."

Christopher Reeve

Chapter 6

Off-the-Shelf Classifiers for Detecting Architectural Tactics

In the previous chapter we presented our novel approach based on a custom made classifier to automatically establish traceability links. Our classifier is built specifically based on the nature of architectural tactics in the source code however there are several off-the-shelf classifiers which could also be used in this context.

Contribution: This chapter presents a ranking comparison for the performance of our tactic detector approach presented in the previous chapter with a number of Off-The-Shelf text categorization methods as well as a voting approach including all the classification methods. Although we evaluated several of the classification techniques in the context of this work, we only report the results for the methods which performed the best and do not include the discussion and the results of the weak classification techniques.

The results are reported in terms of the Recall, Precision and Specificity which all together measure the overall performance of these methods and are commonly used in this context. Our results show that overall, our custom Tactic Detector outperformed the other classifiers and ranked first. In the following sections, the datasets and experiments used to compare and rank classifiers are presented.

6.1 Datasets for Architectural Code Snippets

The initial experiments described in Chapter 5 utilized a small Code Snippets Dataset which was developed to support the task of training and evaluating the tactic-grained classifier. In this chapter in order to extend the scope of the work and rank several classification techniques we increased the number of projects used in the training set from 10 to 50 projects and furthermore instead of five architectural tactics we covered 10 tactics.

For each of the ten targeted tactics, 50 different open-source projects were identified in which the tactic was implemented. For each of these projects we performed an *architectural biopsy* where we retrieved a source file implementing the tactic and one random non-tactical source file. As a result we built a balanced training set from 50 different open source projects.

Each of the code snippets in the training set were selected through one of the following search methods:

- Direct Code Search: Search for the tactic using source code search engines such as Koders. In this search approach we utilized several different keywords commonly used in literature to present and implement these tactics. After retrieving the example of tactical files, a code review was conducted to determine if the file was tactical or not.
- Indirect Code Search: Searching through the meta-data of projects and developers' posts in online forums to find the projects which have implemented architectural tactics. Once a candidate project was selected, the source code review was conducted to identify code snippets implementing tactics and to perform the architectural biopsy.
- Tactics' How-To: Searching through online learning materials and libraries (e.g. MSDN) to find how-to-examples of implementing architectural tactics in a specific language.

6.2 Classification Methods

This section introduces a set of off-the-shelf classification methods chosen to answer the following research question:

- ***RQ6*** *What is the best classification technique for detecting architectural tactics?*

For this purpose, six well known text classification methods have been chosen. The fist step of this experiment includes a preparation phase where all data was preprocessed using standard information retrieval techniques. Terms were stemmed, stop words were removed and the remaining terms were represented as a vector of terms. These vectors of terms were provided as input for each of the classification methods. In each case we justify the reason for including the method in our study.

6.2.1 Tactic Detector

As previously discussed in the chapter 5, the Tactic Detector has two phases of training and classification. The training phase takes a set of preclassified code segments as input, and produces a set of weighted indicator terms that are considered representative of each tactic type. For example, a term such as *priority*, is found more commonly in code related to the *scheduling* tactic than in other kinds of code, and therefore receives a higher weighting with respect to that tactic. During the classification phase the vector of indicator terms is used to predict whether any given source file implements a specific tactic.

6.2.2 Support Vector Machine

A Support Vector Machine (SVM) is a powerful classifier used in various application domains such as bug prediction, text classification and feature selection [74]. SVM selects a small number of critical boundary samples from each class in the training set and builds a linear discriminant function that separates the instances of each classes with a maximum possible separation. When there is no linear separation, the training data is transformed into a higher-dimensional space where it becomes linearly separable. The automatic transformation is done through the technique of "kernel method" [82].

The main property of SVMs that makes them a suitable classifier for our problem is that their ability to learn is independent of the dimensionality of the feature space. SVM based classifier performs well on data with a high dimensional input space which is the case in text classification [74][121]. Typically these methods have been successful in classifying corpii with very many (more

than 10000) features [74]. Particularly this is due to the power of SVMs in avoiding the over fitting problem independent of the number of features. In fact SVMs work according to the margin with which the classes are separated not based on the number of features.

Additionally SVM methods perform well on sparse vectors. This is the case for all text classification problems and is valid for our tactic classification problem in which each java file has only a few non-zero entries.

6.2.3 Classification by Decision Tree (J.48)

Decision Trees (DTs) are a supervised learning method used for classification and regression. The goal is to create a model that predicts the type of a source file by learning simple decision conditions inferred from the words used in tactical files. This decision tree based model is a tree for which internal nodes are test conditions and leaf nodes are categories (tactical/non-tactical). Each internal node of the tree examines an attribute, in our case term frequency. Each branch from a node in the tree examines the value for the attribute.

The attributes which are used to build the tree are chosen based on the information gain theory. It means that the decision tree uses a set of attributes that give maximum information. And the leaf node predicts a category or class [102].

Building a training set for our problem, each file in the training set is represented as vector of terms, and has a categorical type of tactical or non-tactical. The problem is to determine a decision tree that on the basis of word frequencies in each file predicts correctly the value of the category attribute. C45 decision tree generating algorithm is used to induce classification rules in the form of decision trees from a set of given examples.

6.2.4 Bayesian Logistic Regressions (BLR)

One of the effective methods commonly used for the purpose of text classification is Bayesian Logistic Regression model. This method can be applied to problems with a large number of predictor variables, larger than 10000 [56]. Therefore it is suitable for the text classification problem where the dimensionality is high and reducing it can impact the accuracy of the results. In the context of

```
if ( heartbeat <= 0.001961 ) then
{
    if ( monitor <= 0.002667 ) then
    {
        if ( displai <= 0.001825 ) then
        {
            Class2 = "Unrelated"
        }
        elseif ( displai > 0.001825 ) then
        {
            if ( avail <= 0.009456 ) then
            {
                Class2 = "HeartBeat"
            }
            elseif ( avail > 0.009456 ) then
            {
                Class2 = "Unrelated"
            }
        }
    }
    elseif ( monitor > 0.002667 ) then
    {
        Class2 = "HeartBeat"
    }
}
elseif ( heartbeat > 0.001961 ) then
{
    Class2 = "HeartBeat"
}
}
```

FIGURE 6.1: Decision Tree Built to Detect HeartBeat Tactic

text classification, this method provides highly accurate predictions and is generally as effective as classifiers produced by the support vector machine approach.

Utilizing BLR, we aim to learn a classifier, y = f (x), from a set of training examples $D = (x_1, y_1), ..., (x_i, y_i), ..., (x_n, y_n)$. Each document of x_i, is presented in terms of a vector of word frequencies, $xi = [x_{i,1}, ..., x_{i,j}, ..., x_{i,d}]$. The values $yi \in \{+1, -1\}$ are class labels encoding Tactical as (+1) or nonTactical as (-1) of the vector in the category.

In BLR we are interested in conditional probability models of the following form
$p(y = +1|\beta, xi) = \psi(\beta^T xi) = \psi(\sum_j \beta_{jxi,j})$

In what follows we use the logistic link function $\psi(r) = \dfrac{exp(r)}{1 + exp(r)}$

For detecting architectural tactics, $p(y = +1|xi)$ will be an estimate of the probability that the i^{th} source file is tactical. Then a threshold is needed to make decision about this.

6.2.5 AdaBoost

Boosting is an approach in machine learning that creates a highly accurate prediction model by combining many relatively weak and inaccurate ones. The AdaBoost algorithm proposed by Freund and Schapire [50] is one of the most widely applied methods in several domains including software engineering.

Boosting is performed by running a given weak classification algorithm repeatedly over the distribution of the training data and then building an accurate classifier by computing a weighted majority vote of the weak classifiers. The weight assigned to each classifier (in Weka) is equal to *log(1/Beta)wherebeta = error/(1-error)*. AdaBoost increases the weight of cases which are hard to classify at each iteration.

6.2.6 Ensembled Rule Learning: SLIPPER

Another method utilized in this chapter is the rule-based learning algorithm called SLIPPER (for Simple Learner with Iterative Pruning to Produce Error Reduction) [38].

SLIPPER is a standard rule-learning algorithm which is based on confidence-rate boosting and is commonly used for text classification. In this method a weak learner is first boosted to identify a weak hypothesis (an IF-THEN rule), then the training data are re-weighted for the next round of boosting. The main difference of this approach with traditional rule-learning methods is that the data used for learning the rules are not removed from the training set. Instead they are given a lower weight in the next boosting rounds.

The weak hypotheses generated from each round of boosting are merged into a stronger hypothesis in order to ultimately build a binary classifier. The SLIPPER method is recognized as a scalable approach which can work on noisy data with high dimensionality while other divide and conquer rule learning methods suffer from this point. Furthermore, the rule sets generated by this approach are smaller in size in comparison with C45 rules.

6.2.7 Bagging

Bagging, a method described by Breiman [20], works in the same way as boosting but utilizes a simpler way for generating the training set. This approach creates individual classifiers for its ensemble by training each of them on a random redistribution of the training set. Each of the classifiers is trained on a data set created through random sampling of the original training set with replacement. Therefore many of the original items in the training set may be repeated in the resulting training sets while some may be left out. A majority vote on the classification results defines the final results.

Similarly to the boosting case, the training sets are also samples of the original data set, but the "hard to classify" cases more frequently appear in the later training sets. This is mainly because Boosting aims at correctly predicting hard to classify cases.

Bagging is more effective on "unstable" learning algorithms such as decision trees where small changes in the training set result in large changes in predictions [21].

6.3 Tuning Classifiers through N-Fold Cross-Validation

N-fold cross validation experiment is commonly used for tuning the classification methods to find their best thresholds or examining the generalizability of the results when there is a limited number of data points. In our work, we use N-fold cross validation experiments to fine tune each of the classification techniques. We conducted a 5-Fold cross validation experiments over 50 code snippets collected previously. Several rounds of experiments were conducted, where we manually modified the classifiers parameters to obtain the parameters which result in the best accuracy for each of the classifiers. These parameters are reported at (WEBSITE). Tables 6.1 to 6.10 report the outcome of each of the classification techniques in forms of recall, precision and F-Measure on each individual architectural tactic.

The results of the 5-Fold cross validation experiments indicate that SVM classifier was the overall weakest classifier. The remaining 6 classifiers exhibited very close accuracy across several architectural tactics. However, it should be stated that the purpose of these experiments was not to rank the classifier. The 5-fold cross validation experiments were performed to solely tune each classifier,

Table 6.1: 5-Fold Cross-Validation for Audit Tactic

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	
Precision	0.96	0.85	0.85	0.88	0.85	0.94	0.84	Audit
Recall	0.46	0.78	0.85	0.88	0.85	0.91	0.92	
F-Measure	0.62	0.81	0.85	0.88	0.85	0.92	0.88	

Table 6.2: 5-Fold Cross-Validation for Authenticate Tactic

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	
Precision	0.91	0.96	0.98	1.00	0.98	1.00	0.96	authenticate
Recall	0.58	0.94	0.92	0.92	0.94	0.80	0.98	
F-Measure	0.71	0.95	0.95	0.96	0.96	0.89	0.97	

Table 6.3: 5-Fold Cross-Validation for HeartBeat Tactic

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	
Precision	0.91	0.84	0.77	0.89	0.91	0.92	0.77	HeartBeat
Recall	0.62	0.84	0.88	0.84	0.86	0.70	0.92	
F-Measure	0.74	0.84	0.82	0.87	0.89	0.80	0.84	

Table 6.4: 5-Fold Cross-Validation for Pooling Tactic

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	
Precision	0.97	0.94	0.94	0.94	0.98	0.94	0.92	Pooling
Recall	0.66	0.96	0.96	0.94	0.96	0.96	0.98	
F-Measure	0.79	0.95	0.95	0.94	0.97	0.95	0.95	

Table 6.5: 5-Fold Cross-Validation for Scheduler Tactic

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	
Precision	0.98	0.88	1.00	1.00	1.00	0.96	0.86	scheduler
Recall	0.88	0.92	0.98	0.98	0.98	0.98	0.88	
F-Measure	0.93	0.90	0.99	0.99	0.99	0.97	0.87	

Table 6.6: 5-Fold Cross-Validation for Asynch Tactic

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	
Precision	0.58	0.96	0.96	0.96	0.96	0.80	0.95	Asynch
Recall	0.92	0.96	0.98	0.98	0.98	0.78	0.84	
F-Measure	0.71	0.96	0.97	0.97	0.97	0.79	0.89	

Table 6.7: 5-Fold Cross-Validation for HMAC Tactic

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	
Precision	0.78	0.96	0.96	0.94	0.96	0.95	0.91	HMAC
Recall	0.76	0.96	1.00	0.98	1.00	0.84	0.82	
F-Measure	0.77	0.96	0.98	0.96	0.98	0.89	0.86	

Table 6.8: 5-Fold Cross-Validation for RBAC Tactic

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	
Precision	0.83	0.91	0.92	0.92	0.92	0.92	0.86	RBAC
Recall	0.60	0.86	0.88	0.88	0.88	0.88	0.88	
F-Measure	0.70	0.89	0.90	0.90	0.90	0.90	0.87	

Table 6.9: 5-Fold Cross-Validation for Session Tactic

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	
Precision	0.63	0.91	0.91	0.91	0.91	0.76	0.91	Session
Recall	0.80	0.98	0.98	0.96	0.98	0.87	1.00	
F-Measure	0.71	0.94	0.94	0.93	0.94	0.81	0.95	

Table 6.10: 5-Fold Cross-Validation for CheckPoint Tactic

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	
Precision	0.62	0.90	0.94	0.94	1.00	1.00	0.94	CheckPoint
Recall	0.42	0.92	0.94	0.94	0.94	0.97	0.94	
F-Measure	0.50	0.91	0.94	0.94	0.97	0.99	0.94	

and examine the individual accuracy of the classifiers, a second experiment was designed to examine how each of the classifiers performed on an independent large scale software project.

6.4 Ranking the Classifiers based on Hadoop Case Study

In the previous section, a 5-fold cross validation experiment was used to identify best parameters for each classifier. Using those parameters, each classification method was used over Apache Hadoop framework, a large scale software project.

The purpose of this experiment was to (i) rank the classifiers based on their performance, (ii) create a majority voting mechanism between the classifiers and compare it with the tactic detector and lastly (iii) make a suggestion about which classifiers should be used in future.

The following subsections report the results of this study for each architectural tactic in Hadoop. In addition to Recall, Precision and F-Measure, Sensitivity is also reported as it shows more details about the power of each classifier in reducing false positive cases which is very important in the utilization of the classifiers in practice.

6.4.1 Audit Trail Tactic

The outcome of each classifier in detecting Audit Trail tactic in Apache Hadoop project is presented in the Table 6.11. In Hadoop, the Tactic Detector (TD) with the F-Measure of 0.83 outperformed the other classifiers. It also performed better than the majority voting. The low accuracy of majority voting method is because most classifiers participating in the voting had a very low accuracy. The high sensitivity of tactic detector indicates that not only does it detect the audit trail source files with high accuracy, it also ignores the majority of non-tactical source files.

TABLE 6.11: Classifiers Comparison: Audit Trail Architectural Tactic in Hadoop

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	Voting
Precision	0.08	0.02	0.03	1.00	0.03	0.04	1.00	0.67
Recall	0.29	0.29	0.29	0.29	0.29	0.50	0.71	0.50
F-Measure	0.13	0.04	0.06	0.44	0.06	0.07	0.83	0.57
Sensitivity	0.99	0.95	0.96	1.00	0.96	0.95	1.00	1.00

In practice a classifier with sensitivity of %95 is not a good methods as it means that the developers of a medium scale project like Apache Hadoop would receive over 100 of false positive notifications.

6.4.2 Authentication Tactic

Similarly for Authentication architectural tactic as reported in the table 6.12 the Tactic Detector outperformed the other methods, by achieving the F-Measure of 0.66. In case of Authentication, the tactic detector correctly retrieved 70% of relevant source files while the precision of 61% indicates the high rate of false positive cases.

Table 6.12: Classifiers Comparison: Authenticate Architectural Tactic in Hadoop

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	Voting
Precision	0.14	0.16	0.57	0.58	0.17	0.15	0.61	0.47
Recall	0.52	0.61	0.59	0.56	1.00	0.37	0.70	0.66
F-Measure	0.22	0.26	0.58	0.57	0.30	0.21	0.66	0.55
Sensitivity	0.9473	0.9224	0.9929	0.9935	0.9243	0.9657	0.9929	0.98

Similarly we discovered that majority voting has also failed to achieve the better accuracy. This is mainly because the weaknesses of the classifiers participated in the voting.

6.4.3 HeartBeat Tactic

In case of heartbeat tactic, four of the classifiers achieved a recall of 0.96 and higher. However three out of four exposed a high false positive rate. Tactic Detector has achieved the maximum F-Measure among all the classifiers.

Table 6.13: Classifiers Comparison: HeartBeat Architectural Tactic in Hadoop

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	Voting
Precision	0.07	0.31	0.22	0.50	0.35	0.07	0.66	0.57
Recall	0.11	0.59	1.00	1.00	0.96	0.04	1.00	0.96
F-Measure	0.09	0.41	0.36	0.67	0.51	0.05	0.79	0.71
Sensitivity	0.98	0.98	0.94	0.98	0.97	0.99	0.99	0.99

6.4.4 Resource Pooling Tactic

In case of Resource Pooling tactic, although all the methods achieved similar sensitivity rate, J48, Bagging, AdaBoost and Tactic Detector performed very well achieving a F-Measure of 0.87 to 0.93 and the voter achieved similar performance. SVM method performed well in identifying non-tactical source files, achieving a high precision but it has failed to identify most of the tactical files.

6.4.5 Resource Scheduling Tactic

In case of the Scheduling architectural tactic, the performance of the classifiers was less diverse. Tactic Detector and Voting methods exhibited the highest F-Measure.

Table 6.14: Classifiers Comparison: Resource Pooling Architectural Tactic in Hadoop

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	Voting
Precision	0.71	0.13	0.89	0.88	0.87	0.16	0.88	0.89
Recall	0.11	0.44	0.97	1.00	0.87	0.33	1.00	0.96
F-Measure	0.19	0.20	0.93	0.93	0.87	0.22	0.93	0.92
Sensitivity	1.00	0.83	0.99	0.99	0.99	0.90	0.99	0.99

The scheduling architectural tactic was implemented across several modules of Hadoop and over 87 source files. Therefore the precision of 65% and recall of 70% for example, will result in about 44 false notifications, and 82 true positive cases. Considering the size of Hadoop project, this is a significant enhancement in productivity.

Table 6.15: Classifiers Comparison: Scheduling Architectural Tactic in Hadoop

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	Voting
Precision	0.36	0.65	0.64	0.65	0.66	0.32	0.65	0.68
Recall	0.63	0.20	0.87	0.89	0.77	0.78	0.94	0.89
F-Measure	0.46	0.30	0.74	0.75	0.71	0.46	0.77	0.77
Sensitivity	0.94	0.99	0.97	0.97	0.98	0.91	0.97	0.98

6.4.6 Asynchronous Method Invocation Tactic

In comparison to other architectural tactics, Asynchronous Method Invocation has a different nature where the evidence of tactic existence can likely only be seen structurally. However, in Hadoop, we were able to extract instances of this tactic based on comments about the type of communication.

The results of applying different methods on Apache Hadoop projects shows that in five out of eight methods we achieved similar performance. The Bayesian Logistic Classifier, entirely failed to correctly retrieve a single instance of tactical files.

Table 6.16: Classifiers Comparison: Asynch Architectural Tactic in Hadoop

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	Voting
Precision	1.00	0.19	1.00	1.00	0.82	0.00	1.00	1.00
Recall	0.72	0.44	0.72	0.72	0.50	0.00	0.72	0.72
F-Measure	0.84	0.26	0.84	0.84	0.62	0	0.84	0.84
Sensitivity	1.00	0.98	1.00	1.00	1.00	0.97	1.00	1.00

6.4.7 Hash Based Method Authentication

HMAC exhibited the lowest F-Measure across all architectural tactics. The Tactic Detector with F-Measure of 0.11 listed as the last classifier for HMAC, although it had the highest precision of 0.86. The problem is due to the large false positive rate. The Tactic Detector could correctly identify 6 out of 7 java files implementing HMAC,but it also retrieved about 99 files which had nothing to do with HMAC. Across all the other architectural tactics, Tactic Detector shown to be efficient in decreasing the false positive rates.

Table 6.17: Classifiers Comparison: HMAC Architectural Tactic in Hadoop

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	Voting
Precision	0.09	0.12	0.12	0.12	0.12	0.13	0.06	0.12
Recall	0.63	0.50	0.57	0.57	0.57	0.71	0.86	0.57
F-Measure	0.15	0.19	0.20	0.20	0.20	0.22	0.11	0.20
Sensitivity	0.97	0.98	0.98	0.98	0.98	0.98	0.94	0.98

6.4.8 RBAC Tactic

In case of RBAC architectural tactic, most classification methods, could not achieve even 50% of recall. The Tactic Detector was the only exception with recall of 97%. The precision of Tactic Detector, indicates that out of all the retrieved cases about one third are correctly classified java files.

For example in the Hadoop case study, about 121 java files are identified as RBAC of which about 38 of them are correctly classified files, 1 file has been misclassified as unrelated and 83 files are false positives. These numbers are considered the best in terms of accuracy across other classifiers.

Table 6.18: Classifiers Comparison: RBAC Architectural Tactic in Hadoop

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	Voting
Precision	0.12	0.19	0.20	0.42	0.35	0.03	0.31	0.69
Recall	0.13	0.49	0.21	0.21	0.28	0.13	0.97	0.23
F-Measure	0.12	0.27	0.20	0.28	0.31	0.05	0.48	0.35
Sensitivity	0.98	0.95	0.98	0.99	0.99	0.90	0.95	1.00

6.4.9 Secure Session Management

The performance comparison of the classifiers for secure session tactics, shows that SVM and Bayesian classifiers have incorrectly classified most of the tactical files. Slipper classifier ranked first among all the classifiers, then Bagging is the second best classifier. Tactic Detector, AdaBoost, J48 and even the Voter all performed in a similar manner.

In comparison with the other architectural tactics, this is the first time that Slipper classifier wins the competition. In most previous cases tactic detector had performed better than the rest.

Table 6.19: Classifiers Comparison: Secure Session Architectural Tactic in Hadoop

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	Voting
Precision	0.07	1.00	0.84	**0.84**	0.84	0.09	0.84	0.84
Recall	0.11	1.00	0.84	**1.00**	0.84	0.31	0.84	1.00
F-Measure	0.09	1.00	0.84	0.91	0.84	0.14	0.84	0.91
Sensitivity	0.98	1.00	1.00	1.00	1.00	0.97	1.00	1.00

Table 6.20: Classifiers Comparison: CheckPoint Session Architectural Tactic in Hadoop

	SVM	Slipper	J48	Bagging	AdaBoost	Bayesian	TD	Voting
Precision	0.29	1.00	1.00	1.00	1.00	0.12	1.00	1.00
Recall	0.35	1.00	0.94	0.94	0.97	0.68	1.00	0.94
F-Measure	0.32	1.00	0.97	0.97	0.99	0.21	1.00	0.97
Sensitivity	0.98	1.00	1.00	1.00	1.00	0.90	1.00	1.00

6.4.10 CheckPoint Architectural Tactic

The last architectural tactic we studied was CheckPoint, in which the Slipper classifier performed as well as the Tactic Detector. Although J48, Bagging, AdaBoost and Voting classifiers also achieved a very close F-Measure of 0.97 and higher. SVM and Bayesian performed worst among all the methods.

6.5 Examining Research Questions

This section, examines the following research question identified for this part of the research:

- ***RQ6*** *What is the best classification technique for detecting architectural tactics?*

In this work, for which dataset creation and validation is extremely time consuming, we decided to focus on evaluating the practicality of the classifiers in the context that they are going to be used. The results of Hadoop study indicates that the Tactic Detector outperformed the voting approach and this is mainly due to the problem of weak voters.

Furthermore, the purpose of this work is to port a classifier into a Integrated Development Environment such as Eclipse IDE. Therefore first we need to select a classifier which is *reliable*, *easy to use* , relatively *stable* and does not require a heavy involvement of the developers or the actual user to tune or make use of it.

TABLE 6.21: F-Measure Reported for Different Classifiers in Hadoop Case Study

	Audit	auth	heartb	Pooling	sche	Asynch	HMAC	RBAC	Sess	Check
SVM	0.13	0.22	0.09	0.19	0.46	0.84	0.15	0.12	0.09	0.32
Slipper	0.04	0.26	0.41	0.20	0.30	0.26	0.19	0.27	1.00	1.00
J48	0.06	0.58	0.36	0.93	0.74	0.84	0.20	0.20	0.84	0.97
Bagging	0.44	0.57	0.67	0.93	0.75	0.84	0.20	0.28	0.91	0.97
AdaBoost	0.06	0.30	0.51	0.87	0.71	0.62	0.20	0.31	0.84	0.99
Bayesian	0.07	0.21	0.05	0.22	0.46	0.00	0.22	0.05	0.14	0.21
TD	0.83	0.66	0.79	0.93	0.77	0.84	0.11	0.48	0.84	1.00
Voting	0.57	0.55	0.71	0.92	0.77	0.84	0.20	0.35	0.91	0.97

Considering this context and the nature of our dataset, we performed a rank comparison of the classifiers. Table 6.21 reports the F-Measure from all the previous experiments in a table and highlights the top classifier. In detecting 10 different architectural tactics from source code, the Tactic Detector, could rank first, in eight datasets out of ten while in 2 cases the Bayesian and Slipper classifiers performed better than the tactic detector.

Further statistical analysis were performed to examine if the differences between the classifiers are statistically significant or not. Table 6.22 shows descriptive statistics of our data points. In this table we report both mean and median for the F-Measure of each classifier. This data point does not follow a normal distribution, therefore we used Friedman ANOVA test which is a non-parametric test for comparing the median of paired samples.

TABLE 6.22: Descriptive Statistics for F-Measure of Different Classification Techniques

Group	N	Mean Rank	Median
SVM	10	2.60	.170381
SLIPPER	10	3.55	.266862
J48	10	4.60	.661641
Bagging	10	5.85	.707120
AdaBoost	10	4.40	.565247
Bayesian	10	2.40	.174968
Tactic Detector	10	6.50	.813725
Voting	10	6.10	.741164
Total	80		.458313

Friedman test (Table 6.23) indicates the there is a statistically significant difference between the performance of all the classifiers used in our study. However this test does not provide any insight about which classifier is better than the other ones.

Since the Tactic Detector exhibited the highest mean and median (Table 6.22) across 10 different architectural tactics in Hadoop, we performed a pairwise comparison between Tactic Detector and the other classification techniques.

TABLE 6.23: Testing Statistically Significance in Medians of Classifiers Performance

Friedman Test's Statistics	
N	10
Chi-Square	30.395
df	7
Asymp. Sig.	.000

To test the statistical significance between performance of Tactic Detector Table 6.24 reports the results of this experiment using both non-parametric and parametric tests. Uniformly both tests indicated that Tactic Detector is more accurate than SVM, Slipper, AdaBoost and Bayesian classification technique and this conclusion is statistically significant with a p-value of 0.05. Furthermore Tactic Detector performs better than Bagging and J45 classification methods but this conclusion with a p-value of 0.05 and confidence level of 0.95 is not statistically significant.

Similarly although Tactic Detector performs better than the Voting mechanism, the difference between two is not statistically significant. However Tactic Detector is certainly a better approach to be used than Voting, simply because Voting mechanism utilizes the Tactic Detector as a voter.

Table 6.24: Pairwise Comparison of Classifiers with Tactic Detector

Test	Null Hypothesis	SVM	Slip.	J48	Bag.	AdaB.	Bay.	Voting
Wilcoxon Signed Rank Test	The difference in Medians equals to zero	0.011	0.021	0.069	0.123	0.028	0.007	0.173
Paired Sample T-Test	The difference in Means equals to zero	0.001	0.012	0.103	0.154	0.044	0.000	0.193

In terms of stability of the approach, the performance of Tactic Detector has not drastically changed from a cross-validation experiment to a large scale experiment on unseen data of Hadoop. In all the other classifiers, there have been a major change in false positive notifications over Hadoop project. This questions the stability of the methods, across different learning and testing set sizes which can quite often occur in the software engineering context. In many cases we may need to detect architectural tactics in projects of different sizes and also with the different ratios of correct and incorrect classes in the dataset.

Furthermore, approaches such as Bagging, Boosting and Slipper are not simple approaches, they require creation of different rule learners. In contrast, Tactic Detector is a simple, linear classifier which works the same on different dataset sizes. This classifier is easy to use, easy to understand and has now assumptions about the data.

6.6 Summary

This chapter presented several experiments used to rank the performance of a number of Off-the-shelf text categorization methods, tactic detector and a majority voting classifier. The results indicate that over 10 architectural tactics, our approached performed as a top classifier.

We ran two types of experiment, first a set of 5-fold cross-validation experiments, and secondly an experiment on unseen data of Apache Hadoop. While most of the classification methods performed well on 5-fold cross validation, their accuracy significantly dropped over Hadoop case study, mainly due to a large number of false positive cases.

Part III

Traceability for Architecture Erosion

"Treat people as if they were what they ought to be and you help them become what they are capable of becoming."

Goethe

Chapter 7

Notifications and Visualization

Previous chapters, described our approach for creating a strategic infrastructure of architecturally-relevant traceability links. In this part of the research we investigated techniques for effectively utilizing those links to keep developers informed of relevant architectural decisions. Developing a trace usage technique and integrating it with programming IDEs is important for the simple reason that even when traceability links have been created, practitioners often do not utilize them because of the inaccessibility of traceability links to support daily software engineering tasks. [27]

Contribution: The developed architectural tactic discovery technique is used to establish traceability links. The traceability links then are used for keeping developers informed of underlying architectural concerns so that they can modify the design and code without inadvertently degrading the architectural quality. To achieve this goal, our developed solution involves monitoring architecturally significant classes, and providing timely notifications to developers to keep them informed of underlying architectural decisions related to any classes they may be modifying. The monitoring and notification scheme is build upon Event-based Traceability (EBT) [28–30] which allowed trace users to be registered as listeners for change events to specific artifacts. In this chapter we extend EBT so that all classes mapped to a tTP are monitored by the EBT system, and notifications are displayed within the IDE (integrated development environment) to the current user.

7.1 Usage of Event Based Traceability

In this work we incorporate EBT it into our prototyped infrastructure. Our approach has the following steps:

- The tactic detector is used to classify all the classes in a project and create Tactic-Level traceability links.
- The constructed trace links are represented to developers in the form of recommendations, and they are asked to approve or reject the identified links.
- All the tactical classes, are registered as event initiators to the event server through their subscriber manager.
- The source code monitoring module, monitors all the tasks the developer performs. In case of a change to a class implementing a tactic, this monitoring module publishes an event to the notification module.
- The notification module, informs the developer by sending a message to the warning list of the IDE. If the developer clicks on the message, a traceability visualization panel will open which shows the tTP which the changed file is traced to. The developer will see the tactical architectural decisions, and requirements which could be impacted by his/her change. Additionally he/she can view further design knowledge such as codification of tactic, rationale and the business goals behind the tactical decision.

Figure 7.1 depicts the sequence of these actions. First, all the tactical code snippets are registered with a central coordinator which monitors those code snippets for modifications, and generate informational messages to keep the maintainer informed of underlying architectural decisions.

The visual notifications are integrated with a monitoring platform which monitors the code changes and visually informs the developers of potential architectural decisions behind the code, possibly impacted requirements and business goals. The Event-Based Asynchronous Pattern is used to implement the code monitoring infrastructure. This pattern support implementation/integration with the IDE in various ways. Time consuming tasks such as on demand detection of tactics,

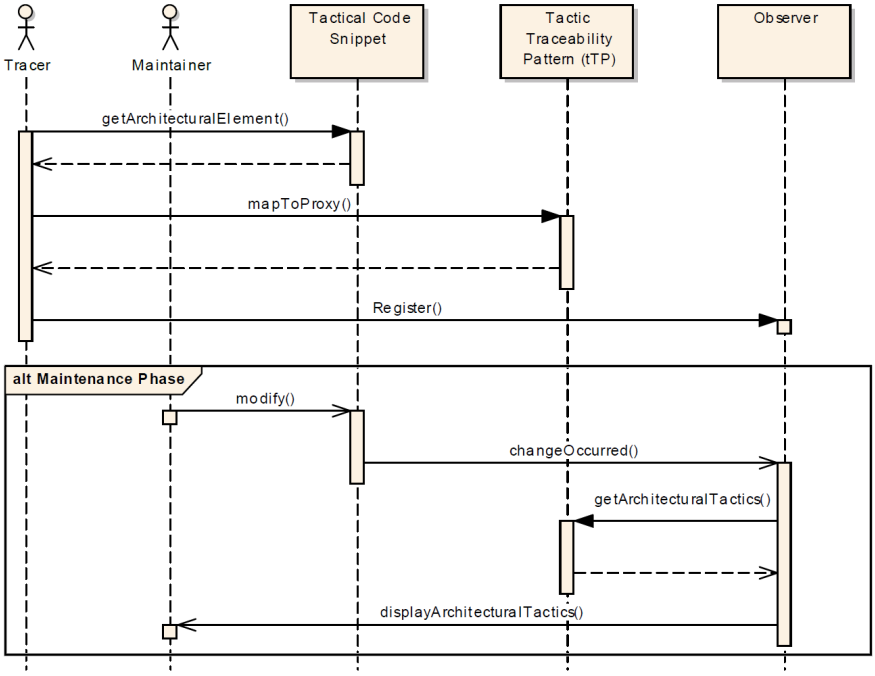


FIGURE 7.1: Monitoring Critical Architectural Element during Maintenance

and visualization of design knowledge are accomplished "in the background", without interrupting programmers activities or slowing down their programming environment. The monitoring infrastructure implemented by this pattern executes multiple operations simultaneously, handle all the activities done by programmers and provide visual notifications when each completes. In the following sections first we present an illustrative example which demonstrates how a tTP can be used in practice. Furthermore we examine the following research question:

> ***RQ7.*** *To what extent can automatically reconstructed unvetted trace links support change notification without inundating developers with excessive false positives?*

7.2 Two Notification Scenarios

To illustrate how notifications and visualizations can be used to address the erosion problem, we demonstrate two examples of actual changes here. The first example uses tTP and EBT infrastructure at the model level, while the second one uses tactic detector at the code level. These two examples are selected to show how we can use the architecture level trace links during daily activities of programmers.

7.2.1 Illustrative Example at Model Level using tTP

To illustrate how tTPs can help mitigate the erosion problem, we provide a step-by-step example of a specific change request executed within the context of Sparx Enterprise Architect (EA). It should be noted that event monitoring and basic notifications are fully functioning in our tool. The actual implementation is in eclipse but could easily be implemented in other environment such as EA. Some of the GUIs depicted in this change scenario are still under development.

For this example we use part of the Lunar Robat Architecture presented in Appendix A.2. In our maintenance scenario, we decide to modify and enhance the robot architecture by adding a high resolution stereo camera to the system to work in conjunction with existing forward looking infrared cameras and laser sensors. The new data is processed by modifying an existing algorithm in the *obstacle detection (OD)* component from the *navigation domain*. The OD component utilizes infrared cameras and other resources to identify obstacles in the planned path. It sends *obstacle* messages to two *Path Planning (PP)* components. However, an underlying architectural decision to use the heartbeat tactic to monitor the availability of the OD component is not explicitly documented, even though the design calls for the OD component to embed heartbeats into the obstacle messages sent to the *PPi* component. This means that obstacle messages must be sent regularly, regardless of whether an obstacle is detected or not.

In the change scenario, the developer starts modifying the OD component to interface with the infrared camera, and to integrate data from the camera into the obstacle detection logic. As the OD component is a monitored element (i.e. registered as playing a role in the heartbeat pattern), an 'architectural' icon is displayed on the screen. This is labeled as (1) in Figure 7.2 and shown as a pyramid over the OD component on the left hand side of the Enterprise Architect (EA) screen. Once an architectural event is detected, the related architectural tactics are visualized in the IDE. In our prototype we display the tactic in a separate pane on the right hand side of the screen (labeled (3)), and color-coordinate specific roles in the primary architectural view (labeled (2))with those of the tactic.

In this way, the developer modifying the OD component is informed of architectural tactics that impact the OD component. The tactical roles of the OD are clearly visible in the visualized tactic, and in this case show that the OD plays the role of a *heartbeat emitter*, and that the heartbeat is

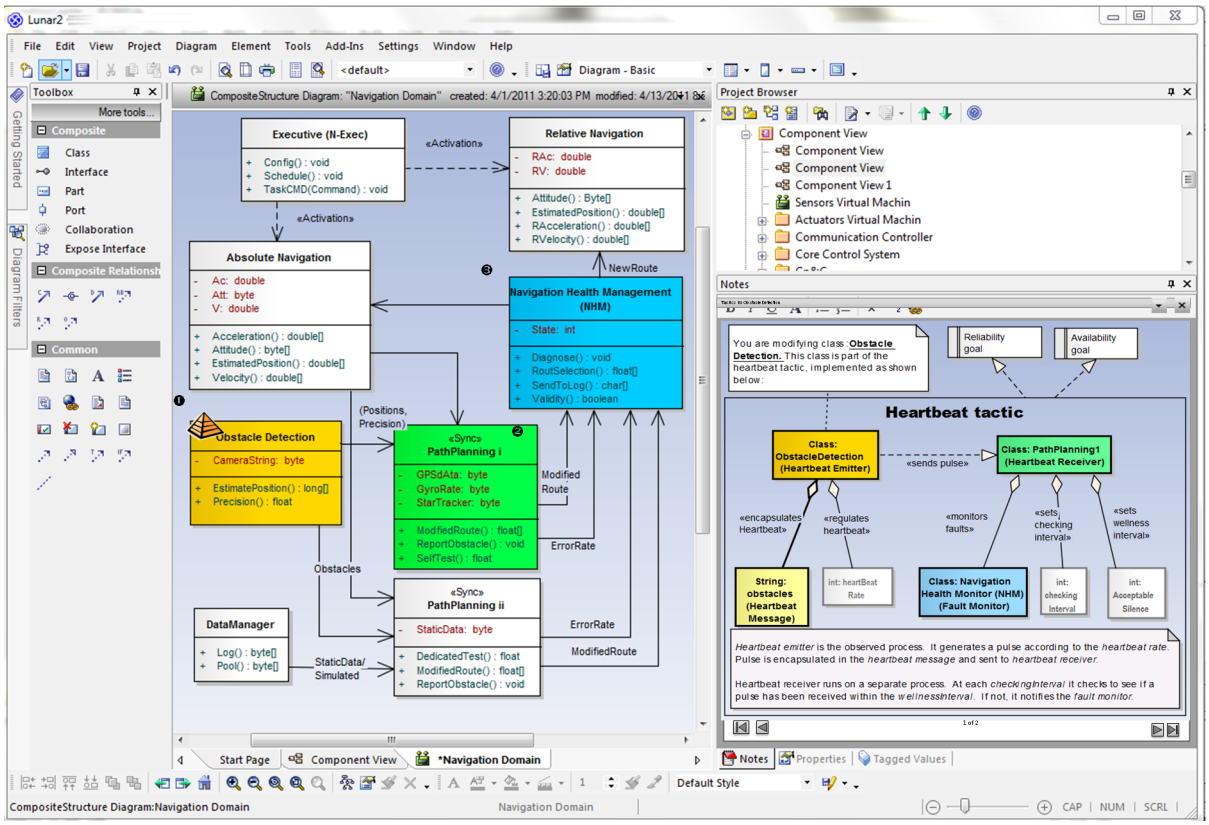


FIGURE 7.2: Visualizing architectural tactics within Enterprise Architect

packaged into the *obstacle* message sent to the *PPi* component. Although the *PPii* component also receives the *obstacle* message, it does not monitor the heartbeat and is therefore not involved in the tactic. This example illustrates how tTPs are used during the maintenance process.

7.2.2 Illustrative Example at the Code Level using Tactic Detector

The second example shows how trace links can be used to help prevent architectural erosion at the code level. We illustrate this with an actual scenario from another project. In TraceLab project [122], there is a change scenario in which a new developer joins the team and starts modifying a part of the system implementing serialization to transmit a copy of the needed data from the shared memory (blackboard) to the executing component. Our traceability framework therefore instigates a "change-impact analysis" request to inform him of the consequence of change. The notification and visualization mechanism in our framework utilizes trace links automatically generated from code to architectural decisions and from architectural decisions to rationale and goals behind the

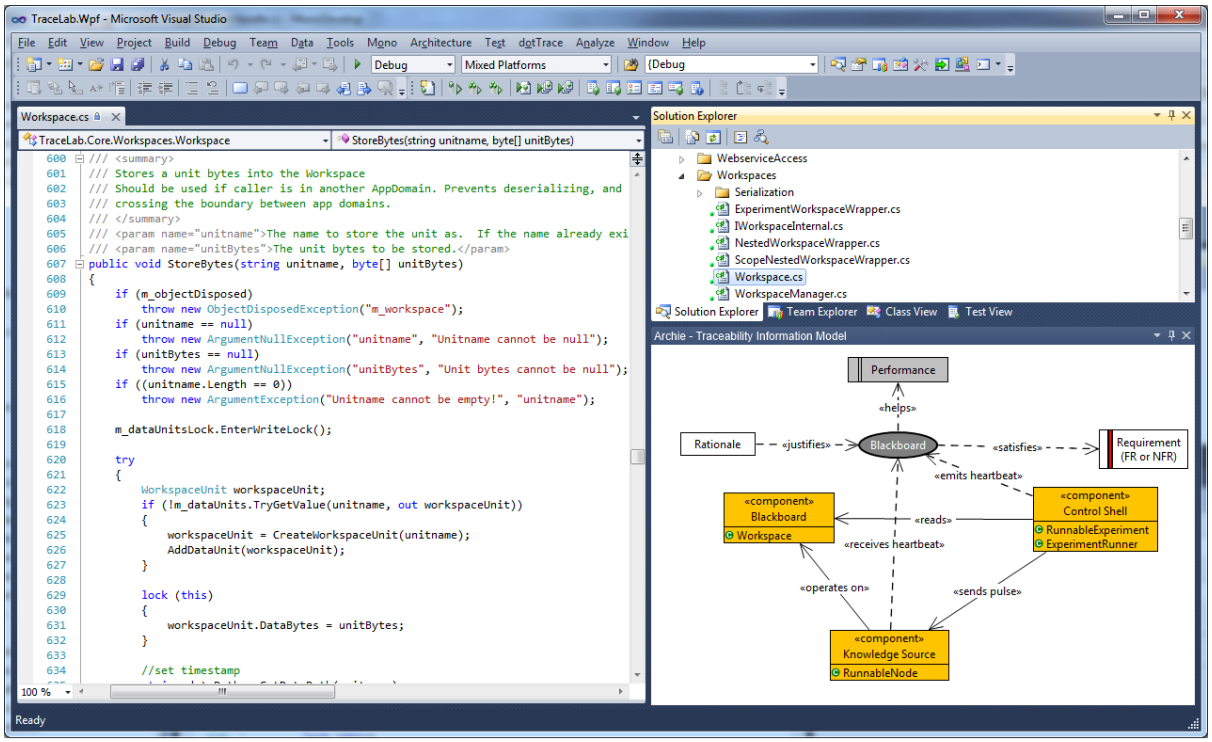


FIGURE 7.3: A Screen Shot of the Archie Tool showing Traceability Established from Implemented Code via the Architectural Decisions to use the Blackboard Pattern to Quality Concerns related to Performance and Usability

serialization code. Figure 7.3 shows parts of our trace infrastructure implemented through our *Archie* tool suite.

Archie introduces a pluggable tool which provides an architectural protection layer for use in a variety of programming IDEs and software modeling tools. Archie utilizes the Tactic Detector to detect and monitor code snippets that implement key architectural decisions in the source code and proactively keep developers informed of underlying architectural decisions during maintenance activities.

In its current form Archie provides support for modeling architectural tactics as tTPs, and for mapping code to elements in the tTP. Mapping can be done manually for all types of decisions and automatically for tactical decisions. Figure 7.3 depicts a screen shot of using Archie to establish trace links by mapping the components implementing Blackboard TIM to its code in the TraceLab project.

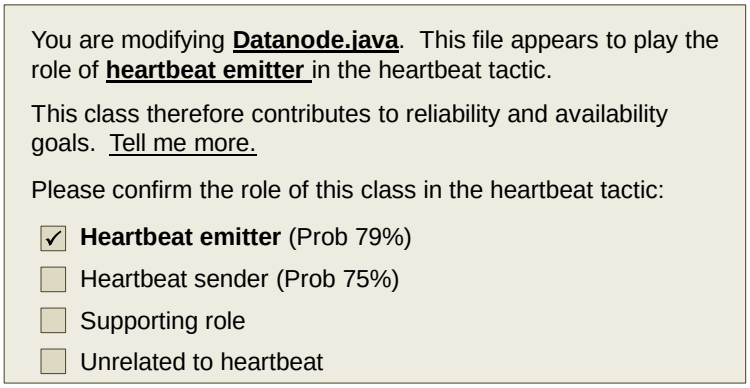


FIGURE 7.4: Utilizing Traces to Generate Maintenance Notifications

7.3 Examining Research Questions

This section, examines the following research question identified for this part of the research:

> ***RQ7.*** *To what extent can automatically reconstructed unvetted trace links support change notification without inundating developers with excessive false positives?*

An important, yet often unexplored research question addresses the issue of whether automatically reconstructed traceability links are good enough for use. Prior work has assumed that the generated (candidate) links must be presented to the user for evaluation and feedback prior to their use [41][40]. We therefore designed an experiment to evaluate the usefulness of the notified messages through coarse-grained traceability links for supporting software maintenance. This task is of particular interest to our work, because of the previously discussed problems of architectural degradation. The experiment utilized the Hadoop change logs for the past four releases, and simulated the scenario in which the generated tactic-level traceability links were used to determine whether a modified class was tactic-related. If it was, we simulated the generation of a message to inform the developer about the underlying architectural tactic. For example, a modification made to the *Datanode.java* class might result in the notification message shown in Figure 7.4 which utilizes traceability to provide useful architectural information.

Table 7.1(a) reports on the number of successfully generated notifications (true positives), the number of unnecessary notifications (false positives), the number of missed notifications (false negatives), and the number of correctly ignored maintenance tasks (true negatives). It also computes recall (the fraction of changes that were tactic-related for which messages were actually sent), precision

TABLE 7.1: Accuracy of Generated Notification Messages during Simulated Modifications to Hadoop

(a) Notification Messages with no User Feedback

	True Pos.	**False Pos.**	**True Neg.**	**False Neg.**	**Recall**	**Prec.**	**Spec.**
Audit	159	5	4405	0	1	0.96	0.99
HeartBeat	256	57	4256	0	1	0.81	0.98
Scheduling	709	1301	2559	0	1	0.35	0.66
Res. Pooling	315	19	4235	0	1	0.94	0.99
Authentication	259	266	4037	7	0.97	0.49	0.93
Averages:					0.99	0.71	0.91

(b) Notification Messages with User Feedback

	True Pos.	**False Pos.**	**True Neg.**	**False Neg.**	**Recall**	**Prec.**	**Spec.**
Audit	159	1	4409	0	1	0.99	0.99
HeartBeat	256	9	4304	0	1	0.96	0.99
Scheduling	709	262	3598	0	1	0.73	0.93
Res. Pooling	315	4	4250	0	1	0.98	0.99
Authentication	259	19	4284	7	0.97	0.93	0.99
Averages:					0.99	0.92	0.98

(the fraction of sent messages that were for tactic-related classes), and specificity (the fraction of changes that were unrelated to any tactics and for which no notifications were sent). Recall of 1.0 was achieved for four of the tactics, and 0.97 for the Authentication tactic. Specificity was over 0.93 in all cases except for the *scheduling* tactic; however precision ranged from 0.35 to 0.96. Table 7.1(b) reports on a second scenario in which we assume that the developer rejects incorrect notification messages, in effect rejecting the underlying traceability link and leading to its removal. Under these circumstances, feedback from initial notifications increases recall, precision, and specificity significantly higher for all five tactics. In fact all metrics are over 0.92 except the precision for the *scheduling* tactic which remains at 0.73.

The results reported for this case study demonstrate that our approach generates tactic-grained links capable of supporting a critical task such as architectural preservation. Therefore the answer to research question of ***RQ7*** is positive; however, it also highlights the importance of capturing relevance feedback from the developers as they receive impact notifications in order to gradually filter out the false-positive links, and ultimately develop a relatively accurate set of traces.

7.4 Summary

In this chapter we utilized previously developed traceability infrastructure and developed a notification & visualization mechanism to help reduce the risk of design erosion. Our approach achieves this by keeping the developers informed of tactical architectural decisions behind the code and notifying them which architectural tactics and consequently software qualities can be affected by the changes they implement. This would address one of the key causes of architectural erosion known as insensitivity to design. The notification mechanism is built primarily on top of automated trace reconstruction technique and has been evaluated in Hadoop case study.

The work presented in this chapter represents an initial attempt to address the challenges introduced by Grady Booch [18] in his IEEE article named "Draw me a picture". In this article, he asks the software engineering community to develop new visualization tools capable of providing greater insights into underlying frictions, design decisions, and social factors of a software system. Our traceability approach makes a partial contribution to addressing this challenge through notifying the developer and showing how different parts of the system work together to achieve various quality goals.

Part IV

Design for Change

You won't get it right the first time anyway!

Frederick P. Brooks

Chapter 8

Variability Points and Design Pattern Usage in Architectural Tactics

In this chapter we conducted a novel analysis of the potential for merging design pattern detection with tactic detection i.e. using one to support the other. We examined various open source software, with the aim of using design pattern detector to increase the accuracy of tactic detector. Our observations indicated that the use of design patterns was sparse (based on the study) and therefore the conclusion was that it would not be worth the cost and effort to incorporate an additional step of design pattern detection into the tactic detection. Nevertheless in the course of conducting the study, we observed that design patterns use in tactics produced some advantages that contribute to our broader goal of preventing architectural erosion. Therefore this chapter aims to utilize a complimentary perspective in preventing architecture erosion. Following David Parnas' notion of "Design for Change"[97][98], we propose a concrete set of guidelines to using design patterns to implement or modify a tactic so that tactic implementation becomes more efficient and tactics become easier to maintain. The guidelines are created through an extensive analysis of hundreds of open source systems; They are presented, through a set of decision trees which support a programmer in his decision making process to select a design pattern based on the constraints or forces he faces in implementing a tactic.

8.1 Implementation Issues of Architectural Tactics

Once a decision is made to utilize a tactic, the developer must generate a concrete plan for the low level design and implementation of the tactic in the code. The code reviews we conducted on various systems showed that, there are various contextual forces, idioms and variability points in each individual tactic which require developers to make numerous decisions in order to implement a tactic. All these variability points, can make implementation of tactics a challenging task, especially for less experienced developers. Figure 8.1 illustrates this point with two concrete examples of developers posting requests for help to online forums because they did not understand how to implement specific tactics. We found many examples of such questions.

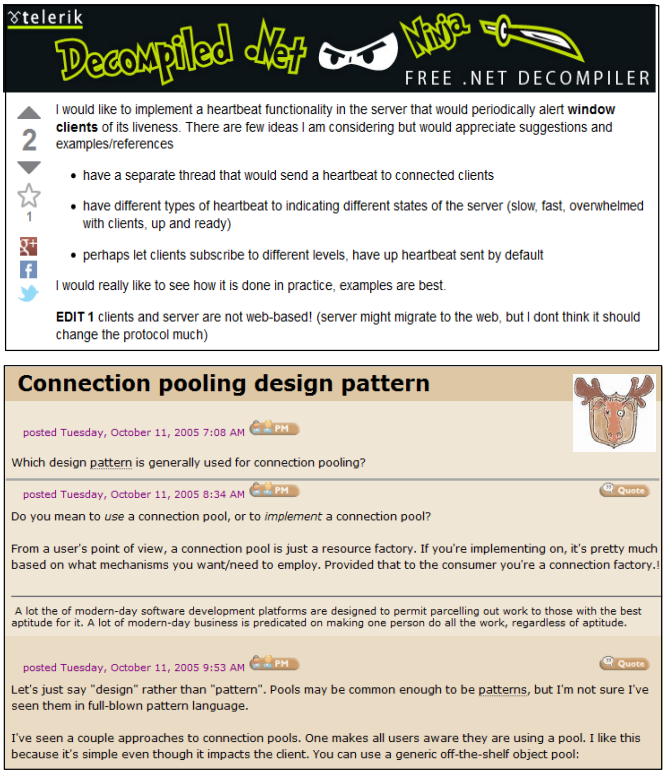


FIGURE 8.1: Developers seek help in online forums to implement architectural tactics

This is a typical knowledge gap that exists between high level architecture design decisions and low level programming decisions, especially for less experienced programmers and unfortunately it has been ignored in many books, materials and tutorial published on software architecture and architectural tactics. If not all, most existing materials focus on high level design and there is not enough guidance for how to implement these tactics specially using well know design patterns.

Unlike design patterns which are described in terms of specific classes and associations, tactics are defined at a higher conceptual level of roles and responsibilities [81] similar to those defined in tTIMs[91]. It is really the responsibility of a programmer to grasp the nuances of the tactic, take into consideration a wide range of project-specific factors that serve as forces upon the tactic, consider various implementation options, and ultimately derive a suitable design solution.

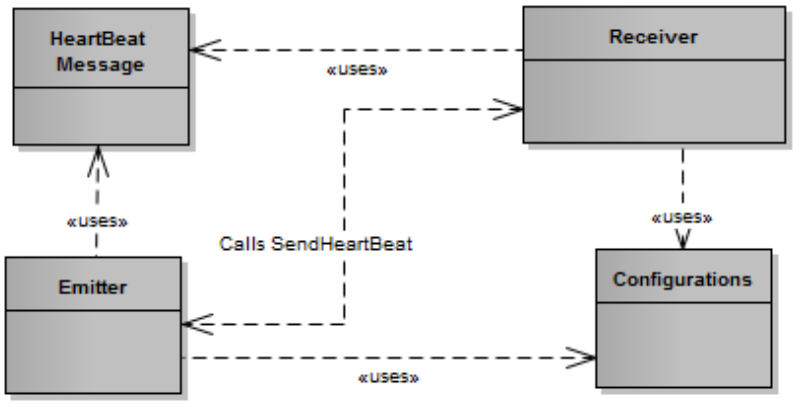
Another important observation in our study of real systems was that usually there is not a single way to implement an architectural tactic. A single architectural tactic could be implemented entirely differently in different systems. Some of these implementations tend to be more efficient, structured, well shaped and well designed and consequently easier to understand and maintain, while there are implementation which are not easy to understand even with a rigorous code review. This increased our interest to investigate the better ways that a developer could implement architectural tactics to make them more suitable for long-time maintenance and evolution.

For example, the heartbeat tactic is a relatively simple tactic used to monitor the availability of a critical component. However, in a previous study of over 20 open source systems [92] we observed numerous variations in how the tactic could be implemented. These included (i) direct communication between the emitter and receiver roles8.2(a), (ii) use of the observer pattern in which the receiver registered as a listener to the emitter 8.2(b), (iii) the decorator pattern in which the heartbeat functionality was added as a wrapper to a core service8.2(c), and finally (iv) in numerous proprietary formats that did not follow any specific design pattern. While an experienced developer can certainly come up with a functioning implementation, it is difficult for any but the most experienced developers to have a complete understanding of all the issues and trade-offs, that might go into making an informed implementation decision.

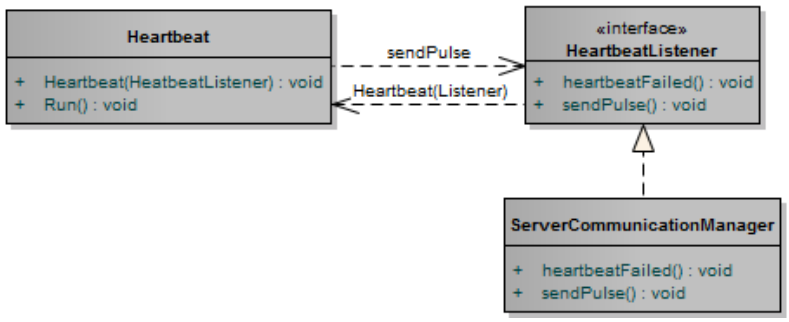
To achieve the envisioned goal for this part of research and answer the following research questions we conducted an extensive study of tactics implementation over a large number of open source systems.

> ***RQ8.*** *Do developers tend to use specific design patterns to implement architectural tactics?*

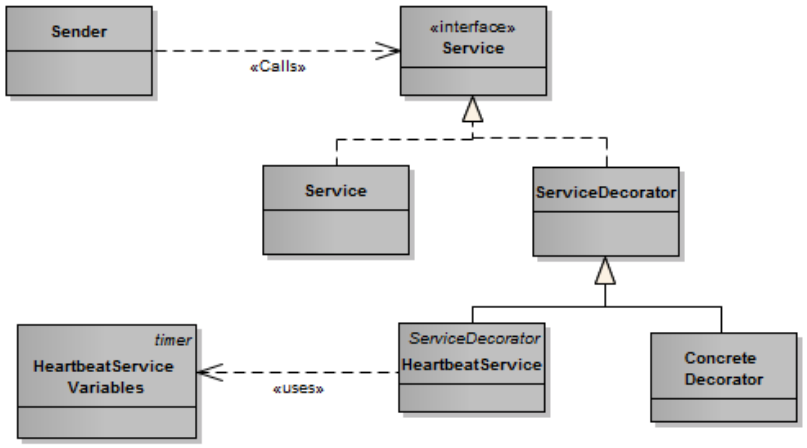
> ***RQ9.*** *What forces influence the choice of design patterns for implementing architectural tactics?*



(a) HeartBeat with configuration files



(b) Observer design pattern to implement the tactic



(c) Decorator design pattern to implement the tactic

FIGURE 8.2: Hadoop : (a)Hadoop, Chat3, smartfrog (b)Amalgam System (c)Thera, JSRB, Rossume Systems

In this study, we investigated the way developers used design patterns within the implementation of architectural tactics (from now on referred to as *tactic/pattern overlaps*). Our approach, findings and examination of research questions are discussed in the following sections.

8.2 Mining Tactic Implementations

Finding a representative sample of tactic/pattern instances is far from trivial. We therefore developed a semi-automated process shown in figure 8.3 for retrieving candidate instances of tactic-related classes implemented using the gang-of-four (GoF) design patterns [51].

This process involves: (i) building a code repository, (ii) detecting and extracting instances of architectural tactics, (iii) detecting and extracting instances of design patterns, (iv) computing the overlap between tactics and design patterns to identify tactic/pattern instances, and lastly (v) manually inspecting the results to remove false positives and to clearly delimit the boundaries of each tactic and each identified pattern. The output from this process provides the raw data needed in the remainder of our study.

(i) Building a code repository The code repository was built by downloading 500 open source projects using Sourcerer [124], which is an automated crawling application designed to extract projects from publicly available open source repositories such as Apache, Java.net, Google Code and Sourceforge.

(ii) Extracting architectural tactics To identify architectural tactics, we utilized our previously developed tactic detection algorithm and tool [93] (Chapter 5). The tactic detector was ran against all 500 previously downloaded projects. Projects were then ranked according to the number of detected tactics, and the top 36 scoring projects were carried forward into the next stage of the analysis.

The final projects are listed in Table 8.1. For each project we report its name, the number of classes in the system, the number of tactic types covered (maximum 13), number of candidate design patterns detected (maximum 20), and the final count of pattern/tactic overlaps as predicted by our automated tools. As depicted in this table, most of the inspected projects provided coverage of 5 or more tactic types; however in order to ensure coverage of all the studied tactics, we included a couple of additional projects simply because they included the targeted tactic type, even though the overall tactic coverage was low.

(iii) Detecting Design Patterns in the Code Pattern detection was performed on the selected projects using an algorithm and tool developed previously by one of our collaborators [107]. The

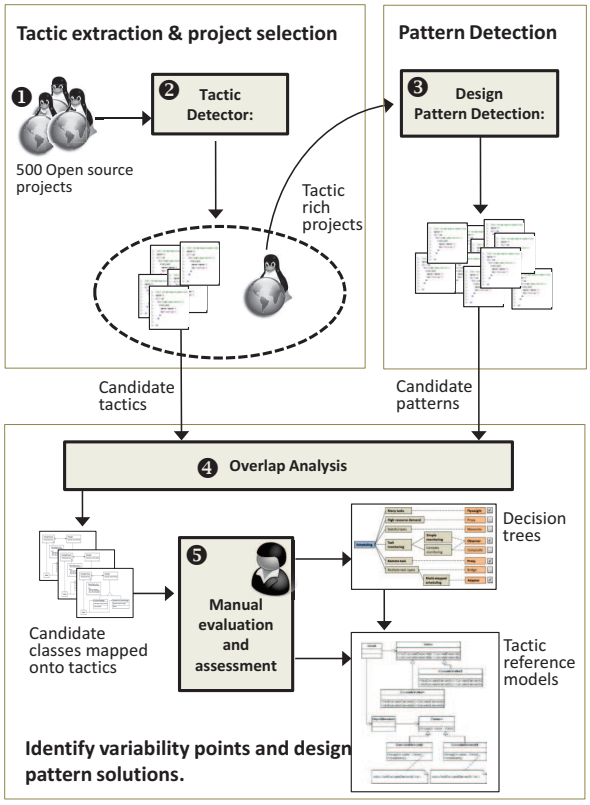


FIGURE 8.3: An Overview of our semi-automated process for mining open source repositories to retrieve samples of tactic/pattern code, identifying tactic-specific variability points, and generating reference models

approach incorporates a catalog of various pattern definitions and applies multiple search technologies during the matching process. The pattern definitions are based on the concept of a feature type, defined as an elementary property of a design pattern likely to recur across different pattern types (e.g., an aggregation relation between two classes or a method return and the instance of the class that contains it). Pattern definitions can be created through composing feature types. A set of 44 feature types were defined and enabled the detection of all 23 so-called Gang of Four (GoF) patterns [51] as well as several of their variants.

Different search technologies have been used which utilize the catalog of pattern definitions and search the source code or reverse engineered class diagram of source code to detect design patterns. The approach has been evaluated and achieved high recall and precision rates of 96-100% for three systems[107].

To detect patterns in the 36 open source projects, we used our existing catalog of pattern definitions

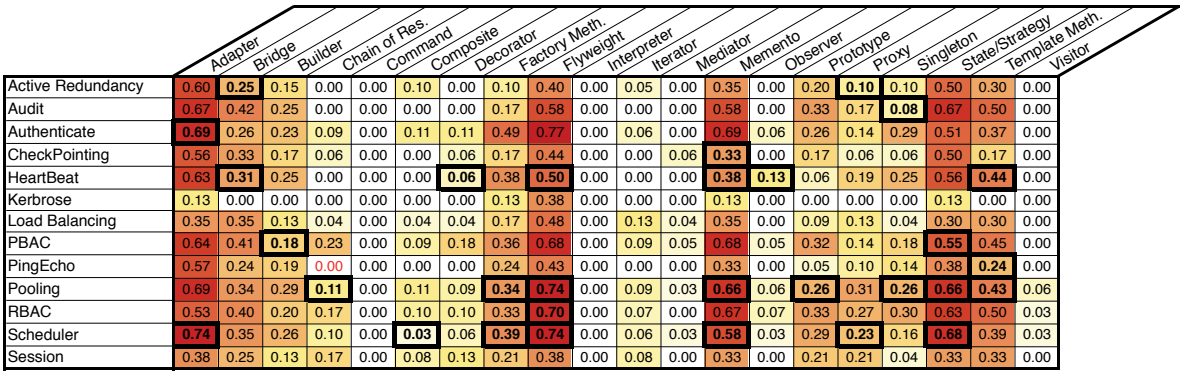
TABLE 8.1: Studied Projects: Size, identified tactics, detected design patterns and observed overlaps

Area	Name	# of classes	Tactic types	Pattern types	Over-laps
Enterprise ERP/CRM	Apache OFBiz	1987	11	14	44
	Apache Tobago	632	9	7	3
	Compiere ERP	1476	10	13	27
	Enterprise	5920	11	3	0
	Jpos	430	7	13	16
	Neogia	3667	11	12	25
	nomadpim	4364	10	18	105
	Open SubSystem	8201	13	17	130
	PaperDog	347	6	7	8
	QuickFix	2291	6	3	1
Frameworks/ Middleware	Apache Cocoon	2945	10	14	62
	Apache DS	1324	9	13	22
	Jboss	5703	13	15	99
	LuntBuild	412	7	7	9
	nomadpim	4364	10	18	105
	open3ds	66	4	1	0
	Oracle CAC	720	10	12	38
	Posit	2545	8	13	22
	vt-middleware	898	9	6	12
Commun-ications etc.	Arsenal-1	3065	9	10	19
	Charliebot	180	8	7	11
	icemud	93	5	3	0
	Projectvianet	1056	9	11	36
Operating Systems	MVctrl	43	2	0	0
	tinyOS	6244	11	16	31
Internet	archive-crawler	757	11	12	38
	Sino	3055	11	14	42
Content Management	Apache Lenya	3661	10	16	65
	mdfiction	227	5	9	11
Distributed Computing	Apache Hadoop	2735	12	14	81
	Apache Triplesec	375	9	9	32
	Mobicents-parlay-ra	2813	6	10	18
	mx4j	891	8	13	40
	OpenJMX	313	7	11	14
	pfc	541	5	13	7
	Spumoni	75	5	10	3

[107]. However due to precision issues we had in detecting similar design patterns, we made the decision to report results from the State and the Strategy pattern together because their structural representation is hard to differentiate. Furthermore we completely omitted results for the Abstract Factory and the Facade pattern as we felt that additional rules and evaluation were needed to detect these correctly. As a result, our pattern detection algorithm was run for 20 GoF design patterns.

Results from the 36 projects are reported in Table 8.1. Column 5, labeled 'Pattern Types', reports the number of distinct design patterns found in each of the projects. Numbers range from zero, meaning that no patterns were detected, to 18, meaning that 18 of the possible 20 pattern types were discovered.

(iv) Computing Overlap The tactic classifier and design pattern detector algorithms were run



	Adapter	Bridge	Builder	Chain of Res.	Command	Composite	Decorator	Factory Meth.	Flyweight	Interpreter	Iterator	Mediator	Memento	Observer	Prototype	Proxy	Singleton	State/Strategy	Template Meth.	Visitor
Active Redundancy	0.60	**0.25**	0.15	0.00	0.00	0.10	0.00	0.10	0.40	0.00	0.05	0.00	0.35	0.00	0.20	**0.10**	0.10	0.50	0.30	0.00
Audit	0.67	0.42	0.25	0.00	0.00	0.00	0.00	0.17	0.58	0.00	0.00	0.00	0.58	0.00	0.33	0.17	**0.08**	0.67	0.50	0.00
Authenticate	**0.69**	0.26	0.23	0.09	0.00	0.11	0.11	0.49	0.77	0.00	0.06	0.00	0.69	0.06	0.26	0.14	0.29	0.51	0.37	0.00
CheckPointing	0.56	0.33	0.17	0.06	0.00	0.00	0.06	0.17	0.44	0.00	0.00	0.06	**0.33**	0.00	0.17	0.06	0.06	0.50	0.17	0.00
HeartBeat	0.63	**0.31**	0.25	0.00	0.00	0.00	**0.06**	0.38	**0.50**	0.00	0.00	0.00	**0.38**	**0.13**	0.06	0.19	0.25	0.56	**0.44**	0.00
Kerbrose	0.13	0.00	0.00	0.00	0.00	0.00	0.00	0.13	0.38	0.00	0.00	0.00	0.13	0.00	0.00	0.00	0.00	0.13	0.00	0.00
Load Balancing	0.35	0.35	0.13	0.04	0.00	0.04	0.04	0.17	0.48	0.00	0.13	0.04	0.35	0.00	0.09	0.13	0.04	0.30	0.30	0.00
PBAC	0.64	0.41	**0.18**	0.23	0.00	0.09	0.18	0.36	0.68	0.00	0.09	0.05	0.68	0.05	0.32	0.14	0.18	**0.55**	0.45	0.00
PingEcho	0.57	0.24	0.19	0.00	0.00	0.00	0.00	0.24	0.43	0.00	0.00	0.00	0.33	0.00	0.05	0.10	0.14	0.38	**0.24**	0.00
Pooling	0.69	0.34	0.29	**0.11**	0.00	0.11	0.09	**0.34**	**0.74**	0.00	0.09	0.03	**0.66**	0.06	**0.26**	0.31	**0.26**	**0.66**	**0.43**	0.06
RBAC	0.53	0.40	0.20	0.17	0.00	0.10	0.10	0.33	**0.70**	0.00	0.07	0.00	0.67	0.07	0.33	0.27	0.30	0.63	0.50	0.03
Scheduler	**0.74**	0.35	0.26	0.10	0.00	**0.03**	0.06	**0.39**	**0.74**	0.00	0.06	0.03	**0.58**	0.03	0.29	**0.23**	0.16	**0.68**	0.39	0.03
Session	0.38	0.25	0.13	0.17	0.00	0.08	0.13	0.21	0.38	0.00	0.08	0.00	0.33	0.00	0.21	0.21	0.04	0.33	0.33	0.00

FIGURE 8.4: Overlaps produced automatically and reported prior to human evaluation

independently across each of the selected 36 projects. The classes detected by the tactic classifier were compared against those detected by the design pattern detector in order to identify tactic/-pattern overlaps. If a pattern and tactic shared at least one class then a candidate overlap was declared. For example, in one case, a decorator design pattern added heartbeat functionality to an http service, and was identified as a decorator/heartbeat instance despite the fact that only a single class overlapped.

The un-vetted results of automatic overlap analysis showing occurrences of each pattern within each of the tactics is reported in Figure 8.4. The numbers in each cell represent the percentage of times a tactic of the given type overlapped with the specified design pattern. For example, the top left hand cell shows that in 60% of the occurrences of the redundancy tactic, there was overlap with a class from the *adapter* pattern. Similarly, the *bridge* pattern overlapped with the redundancy tactic in 25% of its occurrences. It is important to remember that these are raw values computed automatically by our tools and therefore not guaranteed to be accurate until they are evaluated in the next phase.

The color coding of Figure 8.4, shows stronger overlaps in red or orange, and lesser ones in paler colors. It highlights several interesting trends. Some of the design patterns are clearly used more prevalently in tactics than others. For instance *adapter*, *flyweight*, *memento* and *strategy* are the four most commonly occurring patterns. This is may not be really surprising as these patterns contribute to scalability, flexibility, and state restoration, all of which are important in high-performance systems. Furthermore, this could be biased with the imprecision of design pattern detector in detecting these tactics. Unsurprisingly there are no individual tactics that implement a high percentage of patterns.

Column 6 in Table 8.1 shows the number of overlapping tactic/pattern pairs per project. These numbers range from zero, in projects in which no design patterns overlapped with the identified tactics, to 130 in the OpenSubsystem project.

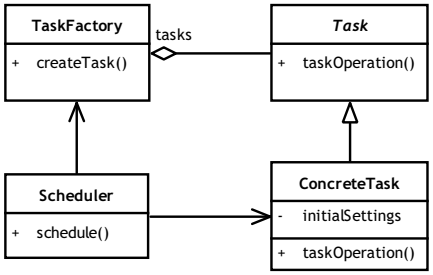
(v) Manual Inspection Given the huge search space of classes in these open source systems, our automated approach significantly reduces the amount of human effort needed to find tactic/pattern overlaps. Nevertheless it is still necessary to manually inspect the results in order to eliminate the incorrectly identified instances and also understand *how* and *why* the pattern was used in the tactic. We tried to extract contextual characteristics of each projects which was forcing utilization of specific patterns in implementation of an architectural tactic.

Identifying correct instances of patterns for deeper analysis was done by initially skimming the pattern instance and eliminating obviously incorrect classifications (i.e. the pattern detector claimed a group of classes to be the heartbeat pattern where clearly they were not). Analysis continued until we found three correct uses of the pattern in the tactic, ran out of cases to evaluate, or searched through 10 tactic/pattern instances without finding a correct use of the pattern. Instances that were not initially ruled out were then checked more carefully to confirm that they represented the claimed pattern and tactic. As part of this process we evaluated approximately 300 instances, and attempted to disperse our efforts equally across the different tactic/pattern pairs. The following steps were then followed:

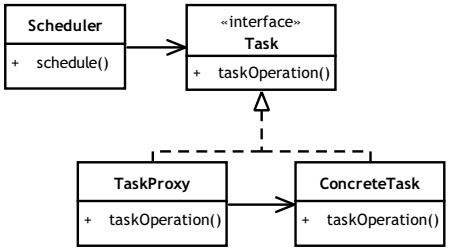
For each correct tactic/pattern instance we evaluated the pattern to answer the question 'what purpose does this pattern play in the tactic?' In cases where the same pattern was implemented for different purposes we studied additional pattern instances until we gained understanding of the multiple contributions the pattern played in the tactic. For each pattern/purpose pair in a tactic, we constructed a generalized class diagram, capturing the classes and their associations needed to implement the design pattern in the context of the tactic. Finally we documented specific factors which might trigger a decision to utilize a specific pattern in the tactic.

On average we found that approximately one tactic/pattern pair was correct for every five evaluated. This is expected, given the precision of the two underlying tools, plus the additional step of combining results [92, 107]. The tactic/pattern overlaps for which we identified correct uses of both the pattern and tactic are depicted in Figure 8.4 with a bold border.

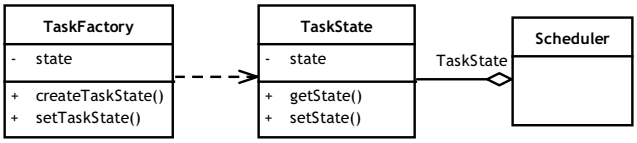
The following sections reports on three different tactic types, and the design patterns that were used to implement them. These tactics are *scheduling*, *resource pooling*, and *heartbeat*. We do not claim to have identified all possible patterns that could be used to implement these tactics, after all it is likely that a creative developer could figure out a way to incorporate almost any design pattern in any tactic. However, what we do claim is that we have identified many common uses of design patterns in the studied tactics.



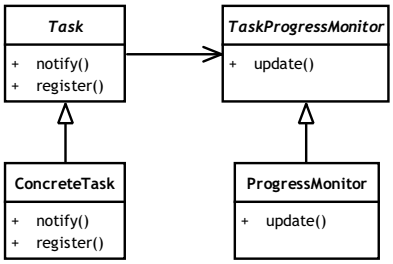
(a) Many Tasks: Flyweight



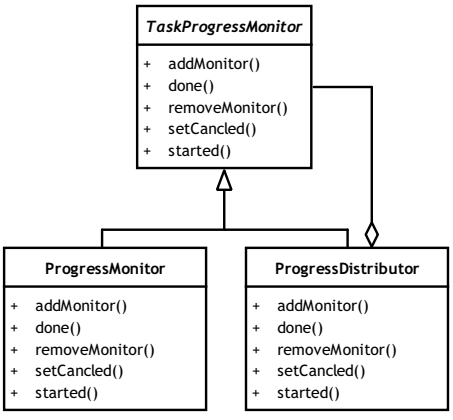
(b) High Resource Demand: Proxy



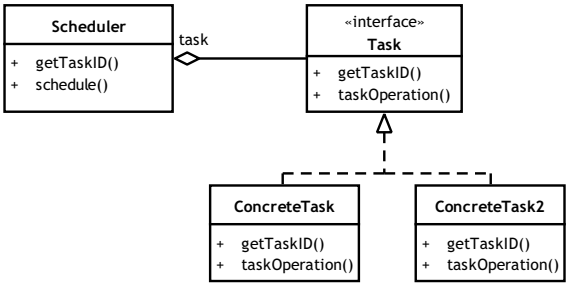
(c) Stateful tasks: Memento



(d) Task Monitoring: Observer



(e) Hierarchy of Monitors: Composite



(f) Many task types: Bridge

FIGURE 8.5: Design Patterns used to address variability points in the Scheduler Tactic

8.3 Scheduling Tactic: Forces and Solutions

The first tactic studied was the *scheduler*. This tactic is commonly adopted to improve system performance in the face of resource contention. It is defined in terms of elements to be scheduled, the scheduler responsible for scheduling them, and a scheduling policy such as FIFO (First in first out), fixed-priority, or dynamic priority scheduling [14, 81, 132]. Bachman et al. identify five independent parameters that must be considered when designing and implementing a scheduler. These include the execution times of units of concurrency, arrival distribution, number of units of concurrency, number of processes, and number of processors. These parameters are used to fine-tune the performance of the scheduler in order to satisfy a response measure such as worst-case latency; however they say nothing about how the scheduler must actually be implemented. As with all design solutions, there are numerous trade-offs to consider. In the case of the scheduler these center around latency, scalability, and code maintainability.

We identified several design patterns which were used quite prevalently to implement the scheduler tactic. These included *adaptor*, *bridge*, *composite*, *flyweight*, *memento*, *observer*, *proxy*, and *strategy*. To more fully understand the purpose of these patterns in the tactic, we analyzed their implementations following the steps outlined previously.

As a result, six primary reasons were identified for adopting design patterns in scheduler implementations. We termed these *variability points* because, in most cases, they could be added as additional features to augment the basic behavior of the tactic. Each *variability point* is described below.

1. Many tasks: If the scheduler is responsible for scheduling a large number of tasks, incurred memory costs may be high and it may be necessary to minimize memory usage. Tasks share intrinsic state such as task priority or task type, while also exhibiting individual properties such as start time, resources required, and so on. The **flyweight** pattern reduces the memory resource requirements, and also reduces time needed to start a task. This concept is generalized in Figure 8.5(a).

2. High resource demand When individual tasks have high resource demands (e.g. high memory), it is more efficient to create them immediately prior to use and destroy them immediately afterwards. The **proxy** pattern can be used so that a proxy object serves as a stand-in for the task while it waits in the queue. The scheduler is unaware that the proxy exists. However, the proxy

creates, invokes, and destroys the task as soon as it is scheduled for execution. This is depicted in Figure 8.5(b).

3. Stateful tasks When the state of the task needs to be preserved between scheduled runs, the **memento** pattern is used to preserve state at the end of a run, and restore previous state at the beginning of the next run. This is depicted in Figure 8.5(c).

4. Task monitoring Tasks often need to be monitored to ensure that they are active and progressing. The **observer pattern** allows a task progress monitor (which could be the scheduler itself) to register as an observer of the task and receive status update notifications. In more complex systems, it may be necessary to have hierarchies of monitors. In this case the **composite pattern** can be used to compose monitors into hierarchies. These uses of patterns are depicted in Figures 8.5(d) and 8.5(e) respectively.

5. Remote tasks The **proxy** pattern can be used to provide a local object as a stand-in for a task that is running remotely. This is not shown in the Figure.

6. Multiple task types When a scheduler is responsible for managing multiple types of tasks i.e. MapTask, ReduceTask, DFSTask, etc, and new types of tasks may be added in the future and/or behavior of those tasks may change over time, then the **bridge** pattern can be used to create a flexible environment in which both the tasks and their behavior can change independently. This is depicted in Figure 8.5(f). When there are multiple types of tasks to be scheduled, and different types of tasks involve different steps (i.e. retrieving data, checking priorities etc) then the **adapter** pattern can be used to adapt each task with the required steps. In this case a generic schedule method is invoked for all tasks and this method is then adapted according to task type.

The identified patterns used in the scheduler are modeled as the decision tree shown in Figure 8.6. A developer needing to implement the *scheduler* tactic could use the decision tree to examine the variability points of the tactic with respect to the specific project.

8.4 Resource Pooling Tactic: Forces and Solutions

The second tactic evaluated was *resource pooling*. This tactic allows limited resources to be shared between clients if neither exclusive nor continual access is needed to the resource. Pooling is typically

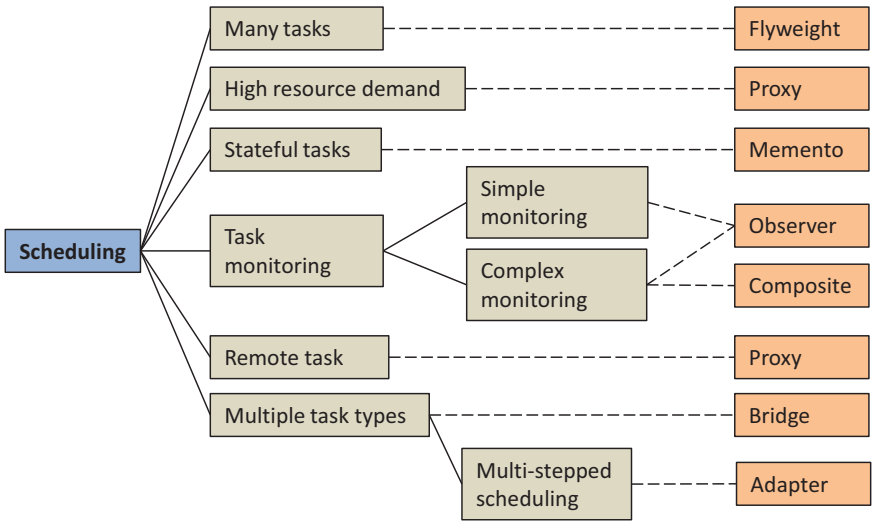


FIGURE 8.6: Decision tree for the scheduler tactic

used to share threads, database connections, sockets, and other such resources [15]. Based on our analysis of resource pooling implementations, we identified several design patterns which were used quite prevalently to implement the tactic. These included *prototype*, *singleton*, *template method*, *strategy*, *factory method*, *chain of responsibility*, and *memento*. The following variability points were identified.

1. Lazy load Shared resources are typically created either at start-up time or upon demand. When the cost of creating a resource in the standard way is high, the **prototype** pattern can be used to efficiently create a clone of a prototypical object.

2. Single point of access Access to a resource pool is managed by a *pooler* role. If many different components need to access the pool there can be significant overhead for establishing and managing a sharing scheme. The problem can be addressed through using **singleton** to ensure that there is only one instance of the pooler at runtime.

3. Multiple kinds of resources When there are multiple types of resources (i.e. thread pooling, connection pooling, session pooling), and if the types are stable, i.e. it is unlikely that new types will be introduced, the **template** method can be used to customize resource management according to resource type. When new types are expected to be introduced the **factory** pattern can be used to create the pools and their associated resources (sometimes referred to as ponds), while the **strategy** pattern can be used to select the appropriate factory method according to type.

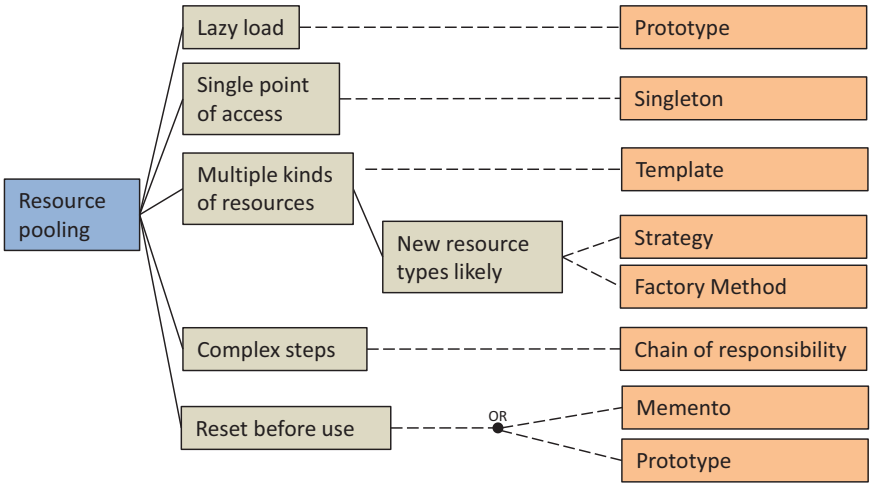


FIGURE 8.7: Decision tree for Resource Pooling

4. Complex steps When the task of creating, managing, and using a resource pool is complex, the **chain of responsibility** pattern can be used to decouple the sender of a request from its receiver. For example a series of requests for checking the existence of the pool, checking the number of objects, or checking the maximum allowed number of objects, can be passed as a command along a chain of objects until the request is handled.

5. Reset before use A resource pool recycles resources. Before a recycled resource can be reused it needs to be reset to its original state. Use either **memento** to reset the resource to its original state, or **prototype** to create a new clone before each use.

These variability points and their associated design solutions are modeled in the decision tree shown in Figure 8.7. However, due to space constraints we do not provide class diagrams for each design pattern.

8.5 Heartbeat Tactic: Forces and Solutions

The third tactic studied was the *heartbeat* tactic. This is used to monitor the availability of a critical component. The monitored component emits a periodic heartbeat message while another component listens for the message. The original component is assumed to have failed if the heartbeat fails [81].

Our study identified six different patterns used to implement the heartbeat. These included *decorator*, *template method*, *observer*, *composite*, *Bridge*, *flyweight*, and *state*. From these we derived the following variability points.

1. Wrap around The heartbeat tactic often needs to be added onto other services or functions which have availability concerns. The **decorator** pattern can be used to wrap those services with heartbeat functionality without the need to modify the services themselves.

2. Piggybacking Sending periodic messages such as a heartbeat can add significant communication overhead and impact system performance. The heartbeat can therefore be piggybacked on other messages such as logging messages used for check-pointing. The **template** pattern can be used to construct a message which carries different kinds of information.

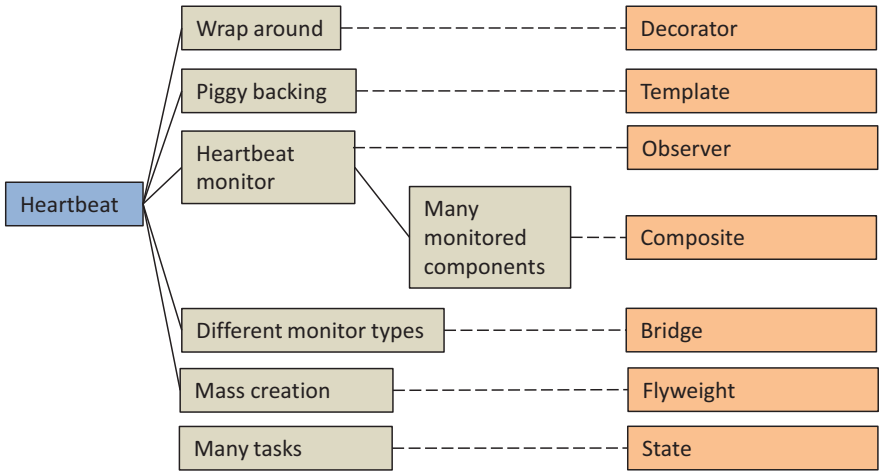


FIGURE 8.8: Decision tree for Heartbeat

3. Heartbeat monitor When multiple components emit heartbeats that need monitoring, the **observer** pattern can be used to register one or more monitors with the heartbeat emitter. In more complex systems where hundreds or thousands of components are being monitored, the **composite** pattern can be used to establish a hierarchy of monitors. In this case a low level monitor checks the health of a group of threads, while its own health is checked by a higher level of monitor.

4. Different monitor types When different types of component are being monitored, the fault monitor must support different monitoring techniques (HTTPS, HTTP, FTP, etc). In this case, the **Bridge** pattern can be used to monitor different types of heartbeat messages, and also check the health of different heartbeat senders in ways that are appropriate for their type.

5. Mass Creation When the heartbeat connection service requires mass creation of the same service specification for many clients, the **Flyweight** pattern can be used to create multiple objects in an efficient way. For example, this approach is appropriate if a HTTP connection is heartbeat enabled in a given project, there are many requests for such connections, and connections and heartbeat services exhibit common properties across all the client requests, then flyweight pattern can be used to create and share multiple HeartbeatHTTP objects.

6. Many tasks: The heartbeat receiver's responsibilities vary based on the status of heartbeat sender. For example the receiver could be in (i) steady-state, i.e. receiving regular messages at predefined intervals from the sender, (ii) compromised-state, when one or more senders is failing to transmit, or in (iii) recovery-state when steps are taken to remediate the problem of the unavailable sender. The *state* pattern can be used to manage these various states.

8.6 a Tactic Reference Model

In sections 8.3 - 8.5 we presented individual patterns implemented in *scheduling*, *resource pooling*, and *heartbeat* patterns respectively. We also presented a decision tree for each of the tactics, depicting the factors driving the adoption of each design pattern.

These factors and their associated implementation solutions can be seen as *variability* points of the tactic. As in a product line, features can be mandatory, optional, or variants. *Mandatory* features represent the essence of the product. In the case of architectural tactics, the core roles, responsibilities, and interactions that define the tactic can be seen as its mandatory features [126]. For example, in the scheduling tactic, there must be a *scheduler* and one or more *schedules*, while in the heartbeat tactic there must minimally be an *emitter* and a *receiver*. *Optional* features are those features which can be added to the product to bring additional value. Finally a *variant* feature is an abstraction of a group of mandatory or optional features, which provides alternate methods of delivering the functionality. In the case of architectural tactics, the identified variability points of the tactic represent a mixture of optional and variant features.

Figure 8.9 shows the reference model we developed for the scheduler tactic. All of the design patterns discussed in section 8.3 are integrated into this model. However, we do not claim that

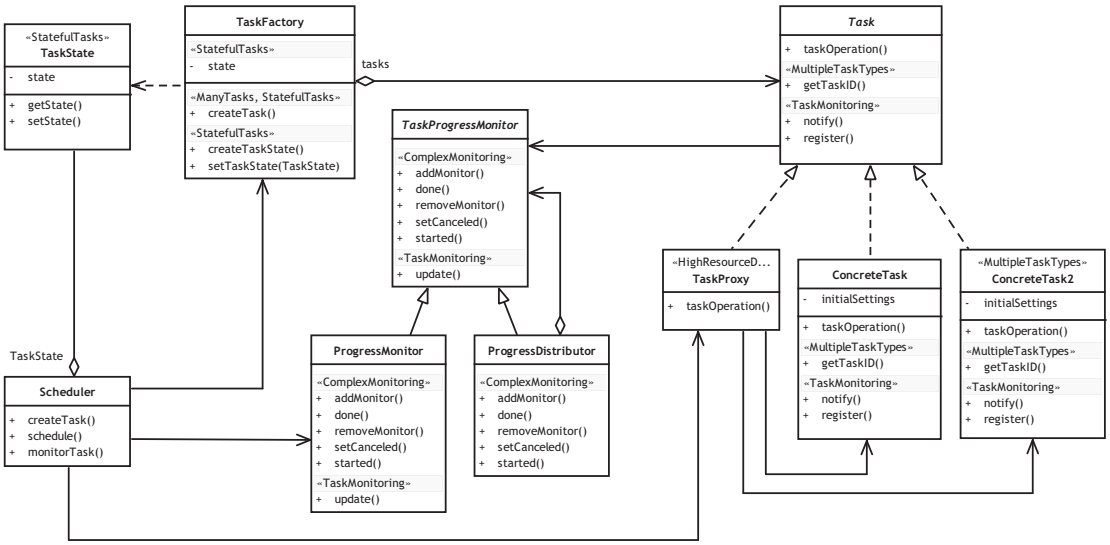


FIGURE 8.9: A reference model for the scheduler tactic. Variability points are marked as stereotypes. These stereotypes are used to reduce the model to deliver only the functionality specified by the user.

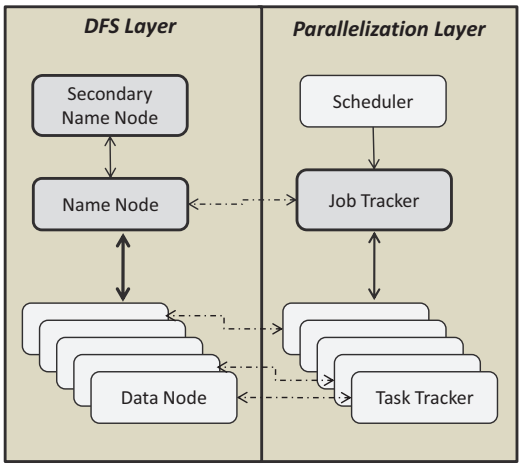


FIGURE 8.10: The high level architecture of the Parallel Computing Infrastructure used in our Case Study

this is the only possible way of combining the design patterns into a complete design solution, nor do we claim that it is the best way. We merely present this solution as a viable design given the variability points of the tactic and their related design patterns.

This reference model can help developers to implement the tactic. Although our longer-term goal is to automate the generation of a customized reference model from a decision tree, we currently provide only the static reference model, where each variability point is labelled with a unique

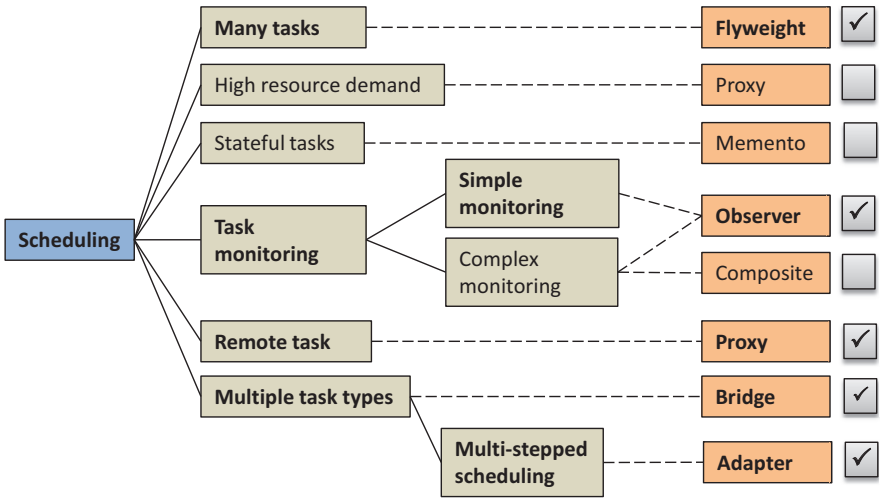


FIGURE 8.11: Desired variability points selected by the developers for the PCI system

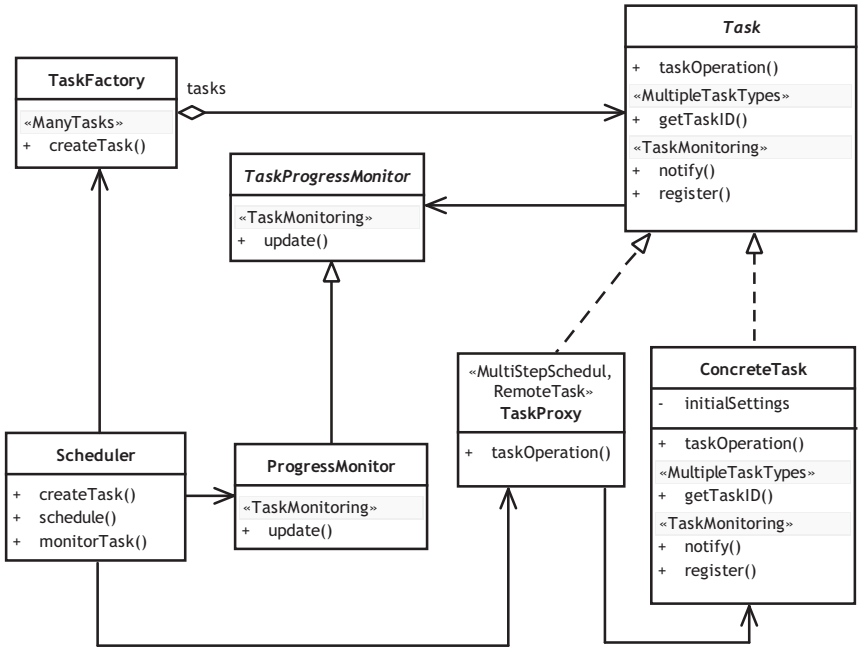


FIGURE 8.12: The reference model modified to retain only desired variability points

stereotypes. This is depicted in Figure 8.9 which shows stereotypes attached to classes, attributes, and to methods. Stereotypes on attributes and methods are shown above the element and refer to all elements underneath. A stereotype on a class means that the whole class with all its attributes and methods contributes to a single variability point and could simply be removed if this variation is not part of the desired configuration. For classes that contribute to multiple variations, the elements inside the class are stereotyped and will be removed if not needed in the desired configuration. If

these reductions result in elements without attributes and methods, then the class itself must be removed and its associations replaced with associations that connect source and target directly. All core elements, e.g., Task, ConcreteTask, taskOperation(), initialSettings, are not stereotyped and will accordingly form the mandatory elements of all configurations, even if all variations are removed.

We present an illustrative example of a system, which we will refer to as *Parallel Computing Infrastructure (PCI)*. PCI is a development environment in which developers can write and/or run parallelized computing tasks. A high-level architectural view is depicted in Figure 8.10. This example is loosely built upon the architecture and implementation of the Hadoop system. PCI has two main layers, the distributed file system (DFS) layer and the parallelization layer. In the DFS layer, *Datanodes* components are responsible for managing and storing data chunks while *NameNode* is a central control point responsible for managing the file system namespace and controlling access by external clients. It keeps track of datachunks managed by *datanode* and also is responsible for distributing replicated data. The parallelization layer is designed to execute tasks through a map reduce paradigm. Operations are submitted by a client and then started by *jobtracker*. *Jobtracker* is also responsible for creating a set of *taskTrackers* to track and report on task status. Job scheduling is a critical function of process parallelization so that the system can achieve high throughput and low response times. The system is expected to handle thousands of concurrent requests.

To explore the possible design space of the scheduler, the architect and/or developers use the scheduler decision tree from section 8.3, and mark the variability points of interest. Their choices are summarized in Figure 8.11. The first key characteristic of the PCI system is it needs to schedule a very large number of tasks. We therefore check off *Many Tasks* to increase parallelization and decrease the performance time. The PCI architecture is cloud-based in which physical redundancy and parallelization are achieved through distributing tasks on different machines. The scheduler therefore needs to manage remote tasks running in different address spaces. We therefore check off *Remote Task*. Scheduling user defined tasks requires customized initialization across the Map Reduce schema, therefore we check *multiple task types* and *multi-stepped scheduling*. Finally we opt for *simple monitoring* of tasks and check *simple monitoring*. Armed with these decisions, the developer utilizes the stereotypes provided in the reference model to remove unwanted parts of the

design and customize the reference model for the PCI project. The end result is shown in Figure 8.12.

8.7 Examining Research Questions

The research questions identified to examine achievement of the automation goal are:

> ***RQ8.*** *Do developers tend to use specific design patterns to implement architectural tactics?*
>
> ***RQ9.*** *What forces influence the choice of design patterns for implementing architectural tactics?*

The results of manual inspection indicated that there is a meaningfull relationship between design pattern usage in each of the tactics. Furthermore, as a result of this inspection we could find the forces impacting each tactic (see sections 8.3, 8.4, 8.5). However we need to conduct further qualitative and quantitative experiments to evaluate the extend each of the design patterns contribute to the implementation of a tactic.

8.8 Summary

This chapter illustrated the results of an extensive study conducted into the use of design patterns for implementing architectural tactics. This study revealed interesting usage trends from which we were able to identify variability points, associate specific design pattern solutions with each variability point, and construct a tactic-specific reference model. In its current form a developer can manually transform the reference model into a customized class diagram showing only the desired features of the tactic. This provides guidance to developers as they implement architectural tactics in a more flexible way and therefore less likely to be eroded over time.

Part V

Conclusion and Summary

"I was born not knowing and have had only a little time to change that here and there."

Richard P. Feynman

Chapter 9

Conclusions

This dissertation has explored the idea of using Software Traceability as a means of preventing the erosion of architectural decisions and degradation of architectural qualities. The work has focused on the new notion of decision-centric-traceability which can potentially address the aforementioned problems through explicitly documenting relationships between quality requirements, architectural decisions, and source code and also using this explicit relationship to perform change impact analysis.

The traceability method developed in this dissertation puts architectural decisions (aka tactics) as the centre of the tracing activity to connect the driving requirements into the implementation source code. Such links later can be used to support change impact analysis and program comprehension during the maintenance process by revealing underlying design decisions behind each code snippets.

Establishing the traceability links requires the recovery and discovery of architectural decisions in the source code. While several other researchers have developed architecture reconstruction techniques for the sake of reverse engineering and design comprehension, this work publishes the first results that utilized such a technique to help developers perform architecture based software development and to strategically maintain critical architectural decisions in the code. First, a hybrid-classifier is utilized to develop an architecture-decision detector, which can be used for regenerating the traceability links. Then the reconstructed traceability links are used to create an architecture protection layer that can be embedded into various modeling environments such as Enterprise Architect, and also within programming Integrated Development Environments (IDEs). The work

presented in this dissertation, automatically detects and monitors architecturally significant classes and when developers modify those classes, it provides timely notifications informing the developers of the relevant underlying architectural decisions. The monitoring and notification scheme is built upon an event-based infrastructure in which all classes related to an architectural decision are monitored, and notifications are displayed to the current user within the IDE.

This chapter summarizes the main findings and conclusions of this dissertation. It also lists several ways in which the work can be extended in the future.

The chapter is organized as follows: Section 9.1 presents a summary of the main results of this thesis, and answers the nine key research questions of this work. Section 9.2 discusses the main threats to the validity of the results as well as the controls that were used to mitigate these threats. Finally, Section 9.3 lists possible extensions and future research directions.

9.1 Summary of Results

The work of this thesis investigates several research questions regarding the utilization of software traceability for partially addressing the problem of architectural erosion. Overall the work is divided into 6 parts. First part focuses on studying several real, large scale software systems and to *Development of a Decision Centric Traceability Method* which can be used for establishing and using the traceability links. The second part utilizes the developed traceability models and aims at *Automating the Construction of the Traceability Links* using a classification technique. Furthermore this part includes several experiments for *Comparing Off-the-Shelf Classifiers with Tactic Detector*. The third part of this dissertation focuses on *Utilization of Traceability Links*. The fourth part of this dissertation goes beyond traceability and aims at *Investigating the Notion of Design for Change for Architectural Tactics* and fighting the problem of erosion with more maintainable implementation of architectural tactics. Lastly this dissertation takes the developed tactic detector technique and focuses on *Building a Smart IDE to Help Developers Address the Design Erosion Problem*.

9.1.1 Development of a Decision Centric Traceability Method

One of the early goals of this dissertation was to conduct an extensive study of architectural decisions in highly dependable and complex systems and understand the important issues in tracing architectural concerns. To our knowledge, this kind of study has not previously been conducted in such a systematic manner, and as a result existing traceability approaches for tracing ASRs and supporting design rationales are not fully practical and tend to suffer from well documented traceability maintenance, and usage problems.

As a result of this study, we proposed the generic decision-centric meta-model to trace quality concerns into design and implementation artifacts. This meta-model recommends that tracing quality concerns should be done through the design decisions into design components and implementation modules. We extended the notion of the meta-model for each specific architectural tactic, and proposed an augmented model called tTP in which, it is clear where to create traceability links in order balance the costs versus benefits of tracing architectural concerns.

Each tTP is centered around a tactic and defines both backward traceability to the driving requirements, quality concerns and rationales of the tactic, and forward traceability to the architectural elements in which it is realized. Furthermore, each tTP defines the internal structure of a tactic in terms of its primary roles and parameters, also relationships between the roles.

The first research question investigated in this part of research was:

> ***RQ1.*** *Does using tTPs potentially reduce the cost and effort of establishing and maintaining traceability links?*

The answer to this research question is positive. The utilization of tTPs provides guidelines for the trace user to trace architectural tactics and at the same time tTPs will result in fewer traceability links.

In practice, a common way of estimating the traceability effort is to count the number of traceability links. Therefore, we conduct an experiment to compare the number of traceability links needed to trace architectural tactics both with and without the use of tTPs. The design artifacts of a Lunar-Robot system was used for this purpose.

Consequently, two different traceability matrices were created. The first matrix, established traceability by using tTPs and mapping architectural elements to tTP proxies. The second matrix established traceability in a more traditional way without using tTPs.

As our results have indicated, tTPs reduce the cost and effort of traceability through providing a set of re-usable traceability links. However, upfront effort is required to create the tTPs. Clearly our approach is constrained by the extent to which similar tactics are reused across projects.

Our observations also have shown that the tTP simplifies the traceability task and therefore reduces the cost and effort of creating a traceability link. For two key reasons, we believe that the results of this case study is sufficient to draw a conclusion. First of all, the Lunar Case study is a real system not a toy example and the tactics covered in this example are realistic example of tactics which might be used in any industrial project. Secondly, the nature of experiment is in a way that, without loss of generality we can expect to see the similar results in any other system.

One main reason for reduction of trace link is that tTPs change the task of link creation to mapping and therefore a set of trace links will be reused and not re-created. The links which are internal to a tTP are reused every time a tactic is traced and the trace user instead of creating all the links just maps the proxy to the design or code elements.

The second research question investigated in this part of research was:

> ***RQ2.*** *How useful are tTPs for notifying the developers of potential erosion through architecture-change impact analysis?*

The traceability links created through tTPs can be used to provide change impact analysis support for the developers during software maintenance. Architecture level change impact analysis can minimize the risk of design erosion.

A set of change scenarios studied over architecture of Lunar Robot Case Study shows that tTPs can be used for proactively keeping developers informed of underlying design decisions when they make changes to the system.

9.1.2 Automating the Construction of the Traceability Links

The first contribution of this work is the introduction of the Tactic Traceability Patterns, however this concepts required manually establishing the traceability links between quality concerns and source code. Therefore chapter 5 presented a technique for automating the reconstruction of traceability links between classes and their related architectural tactics. This automated technique called Tactic Detector and uses a classification method to identify tactical source files and has been used to support the developers stablish the links in each tTPs.

Two different training methods have been used to train the Tactic Detector, the first training method involved using *tactic descriptions* to train the classifier, while the second experiment used actual *code snippets* taken from classes implementing each of the tactics for the training purpose. The underlining assumption is that, establishing a training set based on tactic descriptions is far easier than establishing a training set based on code snippets but the accuracy of the tactic detector tends to be better with code snippets.

For automating the discovery of architectural tactics in the source code and establishing the trace links, the following research questions have been investigated:

> ***RQ3.*** *How accurately does the Tactic Detector generate trace links using two different training methods of tactic descriptions and code snippets? Which one produces better classification results?*

The investigation of this research question shows that both code trained and description based classifiers accurately identify tactic related classes. However the code trained classifier outperforms the description based classifier. Therefore the Tactic Detector method utilized this training mechanism.

> ***RQ4.*** *How effectively can the Tactic Detector identify tactic-related classes for the five targeted tactics in HADOOP?*

The evaluation of this method within the context of the large-scale performance-centric case study of Hadoop system, indicates that this technique is capable of generating fairly accurate links. Furthermore, integrating the concept of tTPs with existing notions of trace retrieval and classification introduces a novel approach to tracing architectural concerns, minimizes the human effort required

to establish traceability, produces traces which can be used to support critical software engineering tasks such as software maintenance, and ultimately helps mitigate the pervasive problem of architectural erosion.

> ***RQ5.*** *How accurately does the Tactic Detector generate role-level trace links for each architectural tactic?*

The hybrid approach to reconstruct role-level trace links did not simultaneously achieve high levels of recall and precision for all the roles in a tactic, and suffered from a large number of false positive cases. Therefore we are not able to have a positive answer for ***RQ5*** and conclude that constructing role-level trace links using our approach was successful.

9.1.3 Comparing Off the Shelf Classifiers with Tactic Detector

In a series of studies we compared Tactic Detector method with a number of Off-the-shelf classification methods. This part of the dissertation, presented a comparison of the performance of our tactic detector approach presented in the previous chapter with a number of Off-The-Shelf text categorization methods. In particular, we evaluate the performance of classifiers from different families of Decision Tree, Support Vector Machines, Bayesian Network. The following research questions were examined:

- ***RQ6*** *What is the best classification technique for detecting architectural tactics?*

The Tactic Detector was recognized as the best classifier among those studied in this dissertation. We reported the results obtained using Recall, Precision, F-measure and Specificity as measures of overall performance. These are commonly used evaluation metrics for such problems. Our results show that overall the Tactic Detector performed better than the other methods, and ranked first. However this was a ranking study based on win/lose cases. A second study was performed to examine if the differences between accuracy of the classifiers are statistically significant or not. The results indicate that Tactic Detector performs better than SVM, SLIPPER, AdaBoost and Bayesian Logistic Regression. The difference between accuracy of the Tactic Detector and these four classifiers is statistically significant.Although Tactic Detector performs better than J48 and Bagging, we need additional data points to conclude that this difference is statistically significant.

Overall simplicity and stability of Tactic Detector make it a better classification technique to be used in integration with a programming IDE.

9.1.4 Trace Link Usage

Previous sections of this dissertation, described our approaches for creating a strategic infrastructure of architecturally-relevant traceability links. In this part of the research we investigated techniques for effectively utilizing those links to keep developers informed of relevant architectural decisions.

Developing a trace usage technique and integrating it with programming IDEs is important for the simple reason that even when traceability links have been created, practitioners often do not utilize them because of the inaccessibility of traceability links to support daily software engineering tasks.

The research question investigated in this section was:

> ***RQ7.*** *To what extent can automatically reconstructed unvetted trace links support change notification without inundating developers with excessive false positives?*

The answer is that the link created using Tactic Detector can be used to effectively keep developers informed of the design decisions behind the code. The low ratio of false positive messages makes the approach more reliable.

The research goal discussed in this section focuses on the task of keeping developers informed of underlying architectural concerns so that they can modify the design and code without inadvertently degrading the architectural quality. To achieve this goal, our proposed solution involved monitoring architecturally significant classes, and providing timely notifications to developers to keep them informed of underlying architectural decisions related to any classes they may be modifying.

To examine the research question ***RQ7.*** the Hadoop change logs for the past four releases were utilized, and simulated the scenario in which the generated tactic-level traceability links were used to determine whether a modified class was tactic-related. The generation of a message were simulated to inform the developer about the underlying architectural tactic.

The results of applying the Tactic Detector over Hadoop projects and utilizing the change logs of Hadoop in four releases demonstrate that our approach generates tactic-grained links capable of supporting a critical task such as architectural preservation.

Therefore the answer to research question of ***RQ7.*** is positive; however, it also highlights the importance of capturing relevance feedback from the developers as they receive impact notifications in order to gradually filter out the false-positive links, and ultimately develop a relatively accurate set of traces.

9.1.5 Design Patterns to Implement Architectural Tactics

In this part of work, a novel experiment was conducted to examine the potentials of using low level programming design patterns to implement architectural design patterns. The following research questions were examined:

> ***RQ8.*** *Do developers tend to use specific design patterns to implement architectural tactics?*
>
> ***RQ9.*** *What forces influence the choice of design patterns for implementing architectural tactics?*

The answer to ***RQ7.*** is positive. The results of a study containing various open source software systems revealed interesting usage trends from which we were able to identify variability points, associate specific design pattern solutions with each variability point, and construct a tactic-specific reference model.

Tactic reference model illustrate various forces which drive the utilization of design patterns for implementing architectural tactics. In its current form a developer can manually transform the reference model into a customized class diagram showing only the desired features of the tactic. This provides guidance to developers as they implement architectural tactics in a more flexible way and therefore less likely to be eroded over time.

9.1.6 Archie: A Smart IDE

The ultimate goal of this work has been the creation of a Smart IDE, capable of supporting the developer to prevent the problem of architecture erosion through increasing their architecture awareness and sensitivity to the design.

Long term goals of the Archie project are to provide a proactive environment to support secure software development though detecting, monitoring and visualizing the critical code snippets in a project, and also preserving the quality of these code snippets through helping developers make better architectural choices. Once completed, we plan to integrate Archie into the Software Assurance Marketplace (SWAMP) www.cosalab.org.

Archie introduces a pluggable tool which provides an architectural protection layer for use in a variety of programming IDEs and software modeling tools. Archie utilizes information retrieval and machine learning techniques to:

- Detect and monitor code snippets that implement key architectural decisions in the source code, including security related ones.
- Proactively keep developers informed of underlying architectural decisions during maintenance activities .
- Automatically trace external architecture specification documents to the source code or design model.
- Perform change impact analysis of architectural concerns at both the code and design level.

9.2 Threats to Validity

This section presents a summary of the main threats to the validity of the results of this thesis, as well as a discussion about the controls that were set in place to mitigate these threats.

9.2.1 Tactic Traceability Patterns

Evaluating a new process is always challenging. Ideally such an evaluation should be conducted within the context of a real project over a period of time as the project progresses from early design decisions to deployment and finally on into the maintenance phase. As this has not been feasible at the current stage of our research, we evaluated the approach through the use of a case study and two quantitative experiments.

The case study of the Lunar Robot created a realistic environment for evaluating the utility of tTPs for preserving architectural qualities, as its architectural decisions are typical of those found in other safety critical software systems. Counting the number of traceability links represents a commonly adopted technique for estimating traceability effort, although given our observations that less effort is required to create traceability links using a tTPs, this approach likely underestimates the realized savings. Furthermore, the maintenance scenarios were constructed using a standard framework, and were therefore representative of a broad spectrum of maintenance tasks. The results from this study therefore suggest that our findings will be applicable across a broad range of systems which utilize some of the architectural tactics we have specified as tTps. The primary threat to validity of our results is that our approach was studied only in a controlled environment, and has not yet been used within the context of a real project. This will be the focus of future work.

9.2.2 Automated Study

There are several threats to validity that may have impacted the automation section of this work. The primary threat is related to the construction of the datasets for training the classifiers. The datasets included 50 open source projects, where from each projects two architectural samples were drawn, one representing tactic-related code and another one unrelated code snippets. The task of locating and retrieving these code snippets was conducted primarily by two members of our research team and was then reviewed by two additional members. This was a very time-consuming task that was completed over the course of three months. The systematic process we followed to find tactic related classes and the careful peer-review process gave us confidence that each of the code snippets was representative of its relevant tactic.

A greater threat to validity is that the search for specific tactics was limited by the preconceived notions of the researchers, and that additional undiscovered tactics existed that used entirely different terminology. However we partially mitigated this risk through locating tactics using direct search and indirect search by looking at projects meta-data and documents. In the case of the Hadoop project, we elicited feedback from Hadoop developers on the open discussion forum. This type of study is always concerned with generalizability of the results. To address this problem we created our initial code-snippets datasets from tactics found in over 50 different open source systems. The leave-one-out and 5-fold cross-validation experiments we conducted are a standard approach for evaluating results when it is difficult to gather larger amounts of data. Furthermore, the Hadoop case study was designed to evaluate the tactic classifier on a large and realistic system. Hadoop has three major sybsystems and many hundreds of programs. We therefore expect it to be representative of a typical software engineering environment, which suggests that it could generalize to a broader set of systems. On the other hand, IR approaches are inherently dependent upon the use of terminology and so there are no guarantees that our classifier will recognize all instances of a particular tactic.

9.2.3 Off-the-Shelf Classifiers

There are several threat to validity related to the classification ranking study we conducted. Although the results indicates that Tactic Detector is the best classifier for detecting tactical code snippets, this is only valid for the studied architectural tactics.

In terms of stability of the approaches, we only evaluated it over a single case study of Apache Hadoop. However this case study included 10 different architectural tactics. The changes in the number of false positive rate for unstable classifiers followed a uniform pattern for all the tactics.

This work only reports the results of the classifiers which performed well in identifying architectural tactic. An initial study contained several classifiers which the best ones were selected to be included in this work.

9.2.4 Design for Change

The tactic/pattern overlap datasets were created using two data-mining techniques, neither of which is 100% precise. This approach introduces potential errors of ommision and commission. Ommission errors were reduced by setting the thresholds in both mining tools quite low in order to favor high recall over high precision. However this also increased the manual evaluation effort which could have resulted in failure to identify instances of correct tactic/pattern implementations. As a result, we clearly cannot claim that we have identified all common cases of design pattern usage in architectural tactics. Instead we claim to have identified many of the common uses. Commission errors were reduced significantly by manually inspecting the resulting overlaps. This task was performed by one author of this paper and validated by the other two. Nevertheless, this is a non-trivial task which sometimes involves deep understanding of the source code. While we are confident that the use of patterns was correctly identified, it is possible that we misinterpreted the intent of using the pattern in a particular context. This danger was somewhat mitigated by examining multiple instances of each pattern use within each of the reported tactics.

Second, our work studies the actual use of design patterns for implementing architectural tactics in open source systems. However, just finding a pattern use does not necessarily mean it was used correctly. Developers may choose less than effective design solutions. To mitigate this problem we only included a pattern in our framework if we found it used in a similar way across three or more systems, or else found its use particularly convincing. Given the nature of open-source development, a pattern that is used in several systems suggests community consensus that it is a solid idea.

Finally, an additional threat to validity arises in the construction of the reference models. These models represent a synthesis of designs found across multiple projects as well as standard knowledge of design patterns available in text books and other material [51]. We have presented our approach as applicable to a broad range of patterns; however due to the time-consuming nature of our study, we have only developed such models for the three tactics discussed in detail in this paper. Based on our study of the other tactics described in this paper, we see no reason to believe this approach is not generalizable.

9.3 Future Work

Automatically detecting parts of a software system that implement tactical architectural decisions can enhance a number of critical software engineering activities, such as change-impact analysis at an architecture level, compliance verification, safety-case construction, program comprehension and prevention of architecture degradation. This work has received the *ACM SIGSOFT Distinguished Paper Award at ICSE 2012* and is currently in the technology-transfer stage at the *US Department of Homeland Security (DHS)*. The automatic discovery of critical code snippets that impact major software qualities, opens new opportunities for novel research, and enables many researchers to conduct deep analysis of the interplay between these critical code snippets and various software characteristics, such as stability of the software, changes, co-changes, and bugs.

In the following several possibilities for the future work is discussed.

9.3.1 Extensions

Seperating Instances of Architectural Tactics: At the current stage of the work, the tactic detector retrieves all the code snippets that implement a particular architectural tactic. In case of some of the architectural tactics, specially the cross-cutting ones there is only one instance of the architectural tactic implemented in a project. However for several of the architectural tactics, there might be one or more instances of the tactic implemented in a project (e.g. Heartbeat). In future work, we aim to incorporate a set of *structural analysis* and *graph theory* techniques to distinguish the different instances of an architectural tactic.

The current implementation of Archie, leaves it to the developers to manually distinguish instances of a tactic and connect then to a tTPs. Being able to distinguish instances of an architectural tactic will reduce developers manual effort and also can provide better insight into the architecture of a software.

Fine-Grained Traceability: This dissertation included several experiments for establishing the coarse-grained traceability links at the tactic level and also fine-grained traceability links to the roles in each architectural tactic. Although the results for coarse-grained links were quite promising, the fine-grained link recovery method did not achieve a reliable accuracy. Therefore, for future work

we aim to utilize more detailed structural analysis and pattern matching techniques to identify the roles each source file plays in implementing an architectural tactic.

Design Patterns and Architectural Tactics In future work we plan to adopt concepts from the feature modeling domain so that each tactic and its variability points will be specified as a feature model from which we can automatically generate a class diagram. We will also extend our work on reference models by adding dynamic views.

In Depth comparison of Several classification Techniques One of the challenges we have faced in comparison of the tactic detector and off-the-shelf classification methods relates to the lack of access to a large number of the datasets. In future, we plan to include 30 more security architectural tactics and conduct a more comprehensive comparison of the classifiers. The current comparison indicates that the tactic detector outperforms the other classifiers but including more data point can help better examine if there is a statistically significant difference between these classifiers or not. To achieve such conclusion a larger and more divers dataset is required.

9.3.2 New Direction

Holistic Software Architecture Reconstruction: Although existing approaches address important aspects of architecture recovery and software program comprehension, they tend to focus only on recovering the structure of software, and fail to discover the complete ground truth architecture. State-of-the-art and state-of-practice tools are currently limited in their ability to retrieve a rich and complete picture of the underlying architecture.

An effective reverse engineering technique should be able to reconstruct the design knowledge generated and consumed during forward engineering of a software product, from early phases of requirements engineering to design and implementation. It is necessary to develop reverse engineering techniques that discover not only the design decisions, but also architecturally significant requirements, business goals, design rationales, various important views of the architecture such as the deployment view and functional view.

This requires using novel methods from various areas of data mining, information retrieval, bioinformatics, principal component regression, code analysis, and structural analysis. More notably, these research trends are beginning to influence the practice of software development in industry.

"Looking back, I have this to regret, that too often when I loved, I did not say so."

David Grayson

Appendix A

Case Studies

This section describes all the case studies used along this proposal.

A.1 Case Study of NASA Crew Exploration Vehicle (CEV)

This section presents the case study of Crew Exploration Vehicle (CEV) system from NASA's Constellation System of Systems. CEV is an exploratory vehicle that is designed to provide round trip transportation for human crews between Earth and space. The CEV is designed to coordinate with transfer stages, landing vehicles, and surface exploration systems in order to support manned voyages to the Moon and beyond. Requirements, architectural decisions, and architectural models of CEV were obtained from a published documentation manual [116] [115].

This case study focuses on the CEV's Guidance, Navigation & Control (GN&C) flight software architecture designed to provide evolving automation, fault detection, isolation and recovery throughout all phases of the flight. The CEV GN&C architecture consists of three major components: the OTM (Onboard Trajectory Manager), the Vehicle Executive, and the core GN&C. The OTM is responsible for trajectory planning. It provides a direct interface for communicating with the crew, and it utilizes a telemetry interface to communicate with ground control and receive activity lists and mode commands. The OTM's primary output is a queue of flight dynamics tasks intended to be executed in sequence by the core GN&C module. This task list is created by the FD Task List Selector (FDTLS) sub-module. The Vehicle Executive is responsible for The OTM and core GN&C

Tactical Decision 1
Quality Goals: Reliability, Availability **Quality Scenario (QS1):** An error in any of the IMUs must not affect the GN&C Flight Software. **Decision (D1):** Voting tactic with self checking. The voting logic helps to ensure data integrity and selection algorithms to quantify and qualify the performance of each redundant unit. **Rationale:** This would guarantee determination of the best performing unit plus the most accurate and reliable result.
Tactical Decision 2
Quality Goal: Reliability, Availability **Quality Scenario (QS2):** The Flight software shall detect any failure in the Chutes/Landing System. **Decision (D2):** Using heartbeat to periodically check the health status of the Chutes/Landing System. **Rationale:** This is an efficient decision for health checking. The availability depends on the health checking period.
Tactical Decision 3
Quality Goal: Reliability, Availability **Quality Scenario (QS3):** Any Failure in GN&C FSW shall not cause Mission Failure or Crew loss. **Decision (D3):** N-Version Programming, GN&C FSW are developed in three different languages by different teams. **Rationale:** This would prevent common mode failures.
Tactical Decision 4
Quality Goal: Performance, Reliability **Quality Scenario (QS4):** The CEV shall perform automated maneuvers and return the crew to Earth during loss of communications with Earth during all phases of the mission. **Decision (D4):** Semantic Based Scheduling with Task sequencing. **Rationale:** This would guarantee real-time accomplishment of both nominal and trajectory missions, task by task based on mission phases and situations.

FIGURE A.1: Tactical Decisions, Rationales, and Driving Requirements in CEV

FSW exchange information via the Vehicle Executive FSW. The Vehicle Executive subsystem is responsible for scheduling and executing all of the vehicle's subsystem executives; e.g., GN&C, power, environmental control, and life support. GN&C is responsible for vehicle navigation during different phases of the mission, and includes primary components of guidance, navigation, control, and targeting processes. GN&C communicates with sensors such as the Inertial Measurement Unit and Effectors such as the Chutes/Landing System through its Subsystem Operating Programs (SOPs). Each GN&C sensor and effector type has its own software subsystem dedicated to receiving raw data from the hardware; FDIR is another component of GN&C which is responsible for Fault Detection, Isolation and Recovery at subsystem level. However health management across different

Vehicle subsystems is performed by the Integrated Health Management System.

The most important architectural decisions, requirements and quality goals in this system are demonstrated in table A.1.

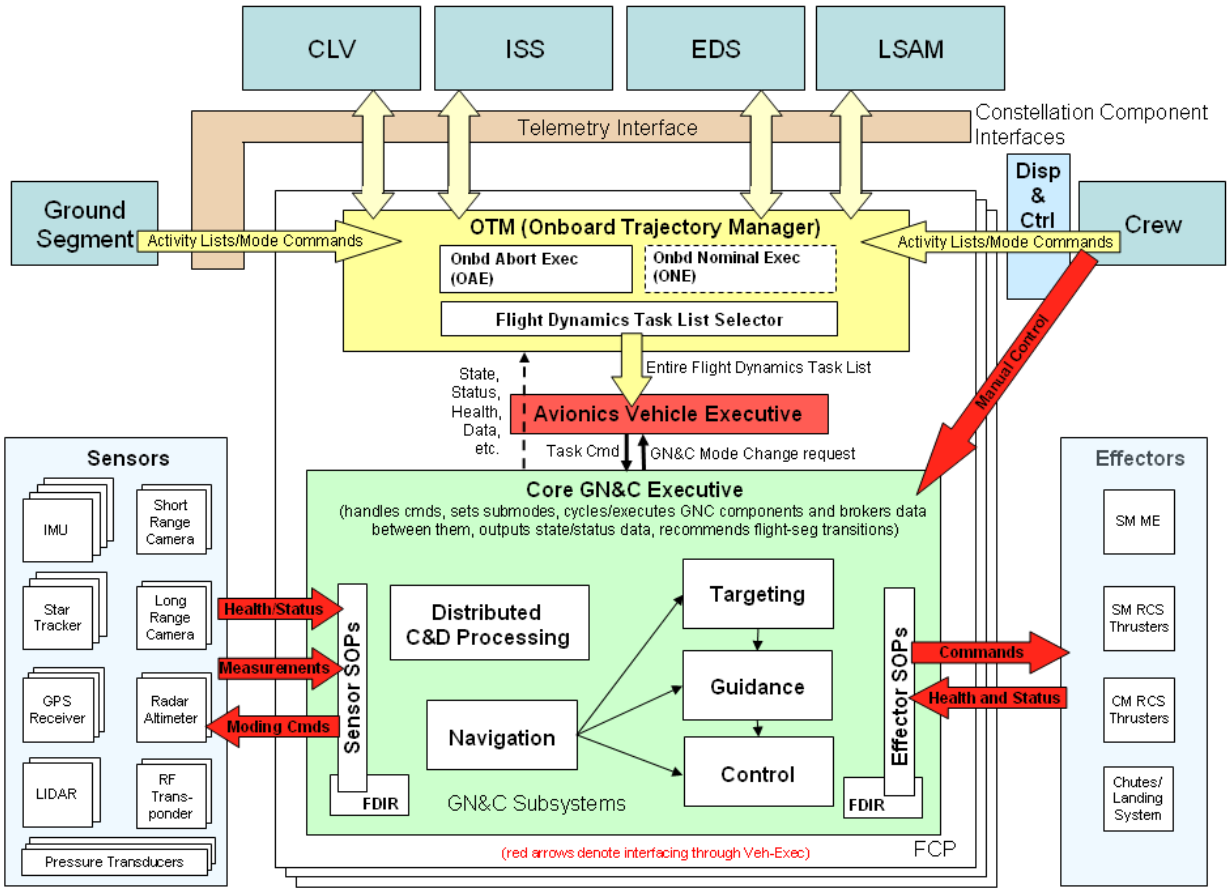


FIGURE A.2: Crew Exploration Vehicle (CEV) system from NASA's Constellation System of Systems

A.2 Case Study of NASA Lunar Robot

The case study of a Lunar Robot system, reconstructed from an extensive set of publicly available NASA documents [70, 95] was conducted to examine the research questions. The robot's primary mission is to autonomously traverse the lunar surface, collect sample data related to comets, dust, and celestial objects, record temperatures, perform scientific experiments, and send results back to the earth-based Mission Control Center (MCC). From a maintenance perspective, our interest is in ensuring that quality concerns, as realized through the various architectural decisions, are maintained throughout the lifetime of the Lunar-Robot software system. Such systems often live far beyond a single lunar mission, as individual components and sometimes entire architectures are often reused in future implementations. Maintenance activities therefore include initial modifications made during the early development phase, modifications made to the system after deployment (which in the case of the lunar robot often means transmitting new software after the robot has been launched into space), and finally modifications made to the system if it is re-used in a new robot system.

A selection of the Lunar Robot's functional and non-functional requirements are shown in Table A.1. An initial analysis of these led to a series of design decisions including the ones depicted in Table A.2. In turn, these decisions resulted in the high-level architectural design shown in Figure A.3. The Lunar-Robot architecture is structured around a *control system (CS)*, *Integrated Vehicle Health Management (IVHM)* system, the *Sensors Virtual Machine* and *Actuators virtual machine*, a *communication system*, and an *operator panel*. Data from the sensors is first passed through the IVHM component for correctness checking and is then forwarded to the CS. The CS uses the data to make decisions and to process high level commands. It then sends lower level commands to the actuators. One of the most interesting components is the integrated vehicle health management system (IVHM) responsible for monitoring the health of the robot, and when necessary, performing dynamic reconfigurations to maintain functionality. The Lunar Robot received inputs from cameras, GPS receivers, rate gyros, and star trackers, and issues commands to mechanical devices such as the power controller, wheels, and scientific instruments.

It is worth noting that the component and connector view of Figure A.3 provides only limited visual clues concerning the architectural decisions behind the design. Certain architectural decisions

are clearly visible, such as the use of partitions to separate different types of functionality. Other decisions are partially visible, such as the decision to include redundant components (given the plural descriptions for GPS receivers and Rate Gyros). However, many of the important architectural decisions listed in Table A.2 are not visible at all, either in this diagram or in lower level diagrams. This lack of visibility is one reason that maintenance efforts often result in architectural degradation.

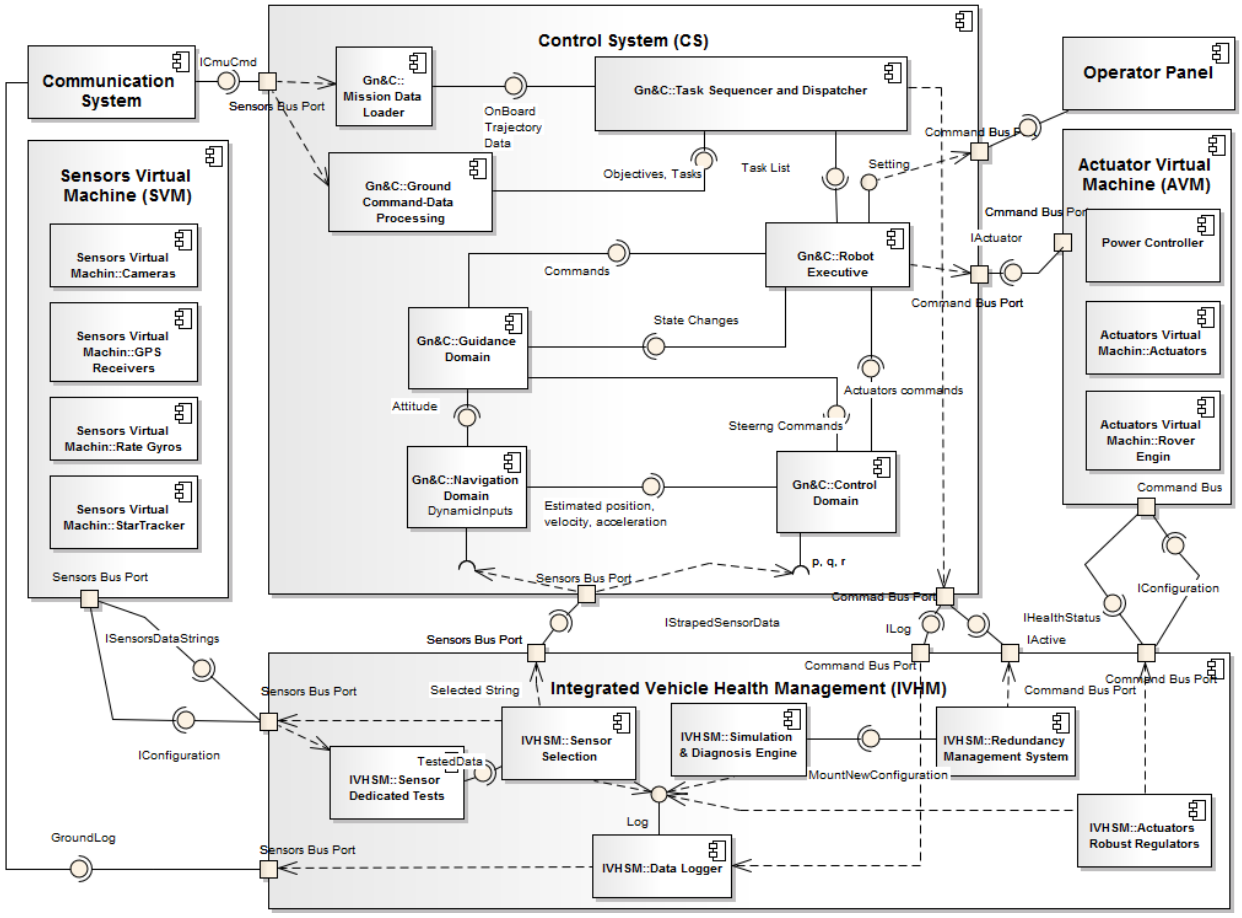


FIGURE A.3: Lunar Robot: High Level Component and Connector View

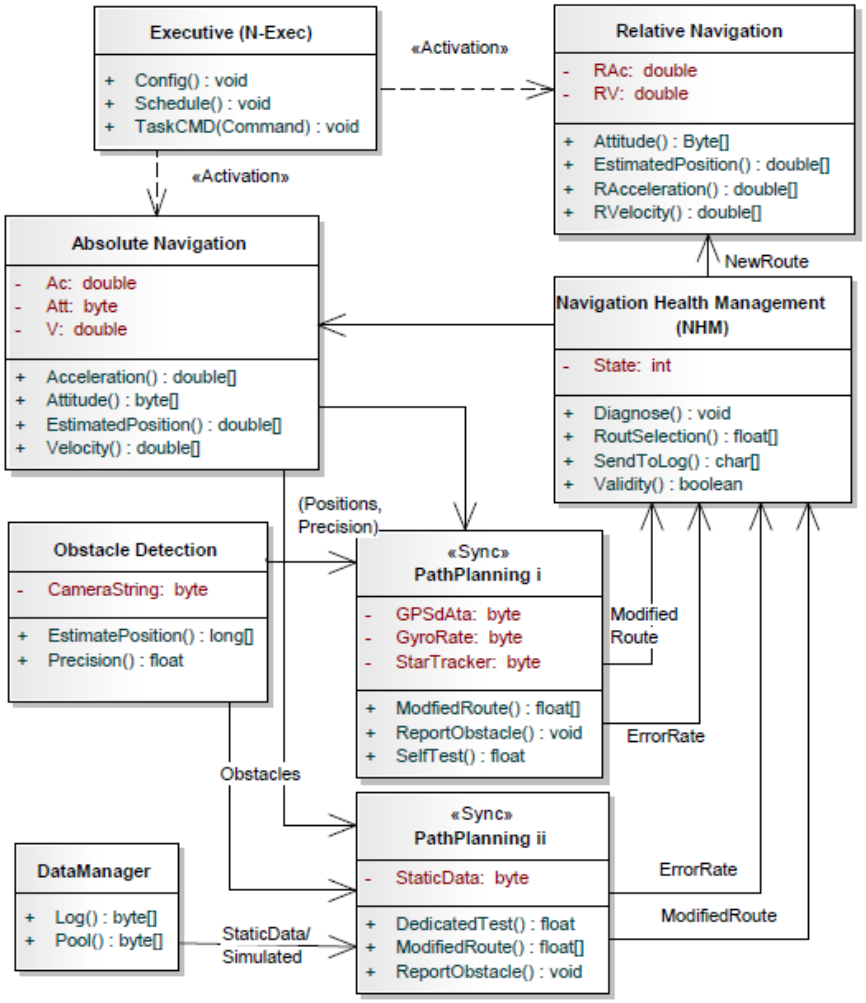


FIGURE A.4: Lunar Robot: Composite Structure Navigation Domain

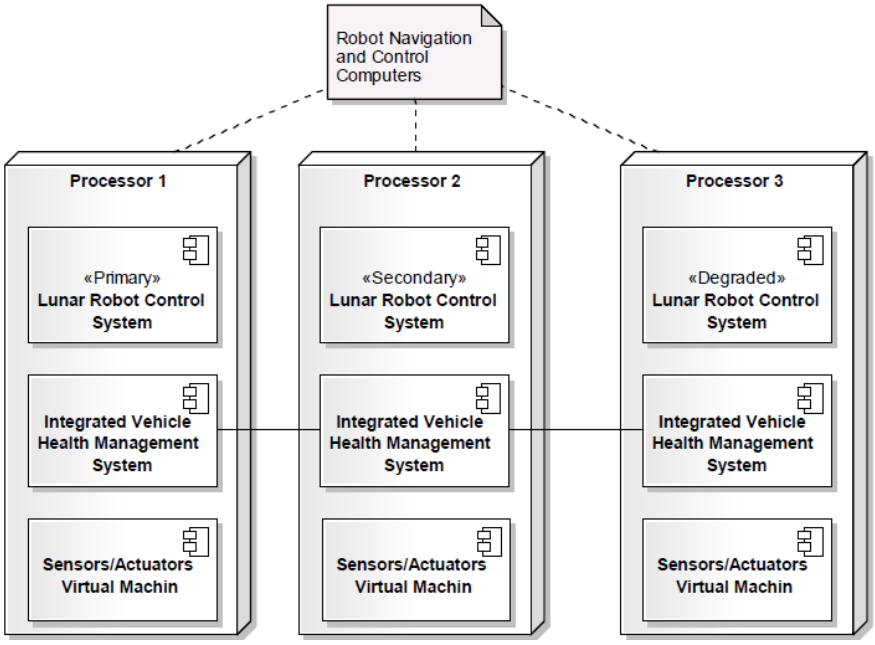


FIGURE A.5: Lunar Robot: Deployment View

TABLE A.1: Lunar Robot: A Sub-set of High-level Requirements

ID	Requirement	Type
1.	The Lunar-Robot (LR) shall have the capability to operate in two modes: manual and autonomous.	Functional
2.	At any time, the LR shall be either inactive, waiting, or active.	Functional
3.	When the LR is inactive, it will check periodically for activation commands.	Functional
4.	The LR shall perform the following functions: move, plan, collect, analyze, record, and transmit functions.	Functional
5.	When the LR is in manual mode it will respond to specific commands received from MCC.	Functional
6.	When the Lunar-Robot is in autonomous mode it will determine and execute the correct steps needed to perform a higher level function defined by MCC. Examples of such functions include reaching a specified location, or gathering atmospheric samples.	Functional
7.	If the Lunar-Robot is in manual mode and and receives no commands from MCC for over *maximum_outOfRange_period* it shall switch to *autonomous_mode* and return to the geographical coordinates of *last_point_of_known_contact*.	Availability
8.	The Lunar-Robot will have a probability of uncorrectable failure of 0.00001 or less during 90 days of its mission on the surface of the moon.	Reliability
9.	Commands sent from MCC to LR will be acknowledged within 3 seconds when communication is feasible.	Responsiveness
10.	Transmissions between MCC and the LR shall be secure.	Security

TABLE A.2: Lunar Robot: Primary Architectural Decisions

ID.	Decision
1.	The Lunar Robot control system will be replicated using active redundancy with graceful degradation. There will be replications.
2.	Each instance of the Guidance, Navigation and Control subsystems(GN&C) will run on a separate process, and will be configured uniquely.
3.	The tasks to be performed at various phases of the mission will be managed in individual configuration files by the robot executive and domain executive modules. The task sequencer reference model is used to achieve real-time task distribution and execution
4.	Task scheduling will be performed using the semantic-based scheduling algorithm. This approach takes priorities of various tasks, and the current mode of operation into consideration. It allows task priorities to be established according to the current operating mode.
5.	Each of the three domains will have a dedicated health management module which will implement two-self checking for each running thread (i.e. two separate threads will run the same task and must be in agreement). In case of a mismatch, the result which most closely matches the data produced by the simulation engine will be used.
6.	Critical modules in each of the Guidance, Navigation and Control (GN&C) domains will be redundantly coded in three different programming languages by three independent teams.
7.	Thread execution will be scheduled according to predefined priorities using preemptive scheduling.
8.	All sensors will be monitored using the heartbeat tactic.
9.	All data received from sensors will be tested for validity i.e. using CRC.
10.	A majority voting schema will be used to select the most accurate data from a set of replicated sensors.
11.	Two separate buses will be used for conveying sensor data and control data respectively. Note: This allows different scheduling strategies to be applied to each bus.

A.3 Case Study of Hadoop Framework

The Hadoop project is one of the Apache Foundation's projects. It is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-avaiability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-availabile service on top of a cluster of computers, each of which may be prone to failures. Hadoop is modeled after Google's MapReduce / GFS framework and is implemented in Java.

Hadoop is *economical*, It is preferable to have more low-performance, low-cost hardware working in parallel than to have less high-performance, high-cost hardware. Also it's *reliable*, it's applicable in distributed systems where failure becomes the norm. With Hadoop design, At some point, the number of discrete systems in a cluster will be greater than the mean time between failures (MTBF) for any hardware platform, no matter how reliable, so fault tolerance must be built into the controlling software.

Major Subsystems:

- **Hadoop Distributed File System (HDFS)**: A distributed file system that provides high-throughput access to application data.
- **Hadoop MapReduce**: A YARN-based system for parallel processing of large data sets.
- **Hadoop YARN**: A framework for job scheduling and cluster resource management.
- **Hadoop Common**: The common utilities that support the other Hadoop modules.

Main Architectural Styles Master-Slave style is dominant across all Hadoop's subsystems. The master-slave architecture is a variant of the main-subroutine architecture style that supports fault tolerance and system reliability. In this architecture, slaves provide replicated services to the master, and the master selects a particular result among slaves by certain selection strategies. The slaves may perform the same functional task by different algorithms and methods or by a totally different functionality.

A.3.1 HDFS Architecture

Hadoop distributed file system is built around two major components:

NameNode: A master server that manages the file system namespace and regulates access to files by clients **DataNodes**: Store the actual data in HDFS files. Usually one per node in the cluster.

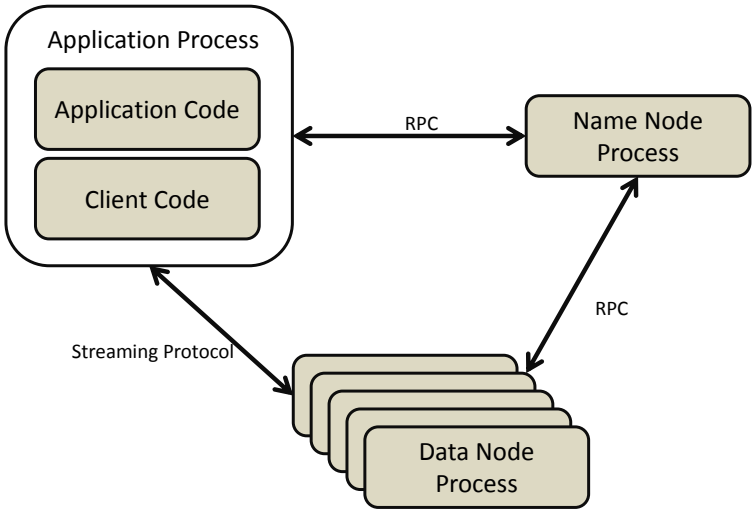


FIGURE A.6: Hadoop Distributed File System: Module View

One of the main goal of HDFS is fast parallel processing of lots of data. To accomplish that it's necessary to have high level of redundancy. Client need to break his/her own data file into smaller "Blocks", and place those blocks on different machines throughout the cluster. Copying the blocks of data on more machines will increase the redundancy level and throughput of parallel operations. Also as these machines may be prone to failure, the data redundancy will guarantee that data loss will not happen. Therefore in HDFS each block will be replicated in the cluster as its loaded. The standard setting for Hadoop is to have (3) copies of each block in the cluster. However this is a configurable parameter (inside hdfs-site.xml).

The copy operation on HDFs has the following workflow (shown in A.7): For each block, the Client consults the Name Node (usually TCP 9000) and receives a list of (3) Data Nodes which should have a copy of the block. Then it's the responsibility of the Client to write the block directly to the Data Node (usually TCP 50010). The replication is done through the receiving Data Node, it will replicate the copied block to (two) other Data Nodes, and the cycle repeats for the remaining blocks. The Name Node will not be in the data path. The Name Node is playing the role of a look

up table or dictionary which only provides the map of where data is and where data should go in the cluster.

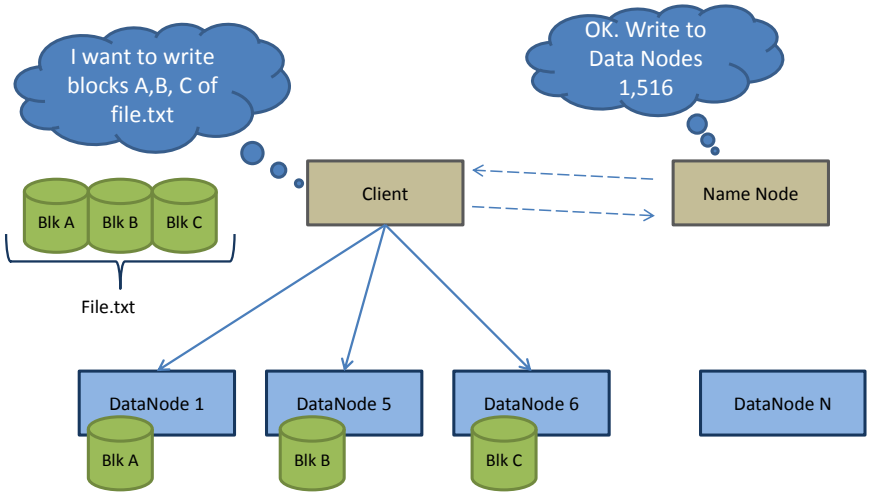


FIGURE A.7: Writing-Files-to-HDFS

protocol The protocol package is used in communication between the client and the namenode and datanode. It describes the messages used between these servers.

security security is used in authenticating access to the files. The security is based on token-based authentication, where the namenode server controls the distribution of access tokens.

server.protocol server.protocol defines the communication between namenode and datanode, and between namenode and balancer.

server.common server.common contains utilities that are used by the namenode, datanode and balancer. Examples are classes containing server-wide constants, utilities, and other logic that is shared among the servers.

client The client contains the logic to access the file system from a user's computer. It interfaces with the datanode and namenode servers using the protocol module.

datanode The datanode is responsible for storing the actual blocks of filesystem data. It receives instructions on which blocks to store from the namenode. It also services the client directly to stream file block contents.

namenode The namenode is responsible for authorizing the user, storing a mapping from filenames to data blocks, and it knows which blocks of data are stored where.

balancer The balancer is a separate server that tells the namenode to move data blocks between datanodes when the load is not evenly balanced among datanodes.

tools The tools package can be used to administer the filesystem, and also contains debugging code.

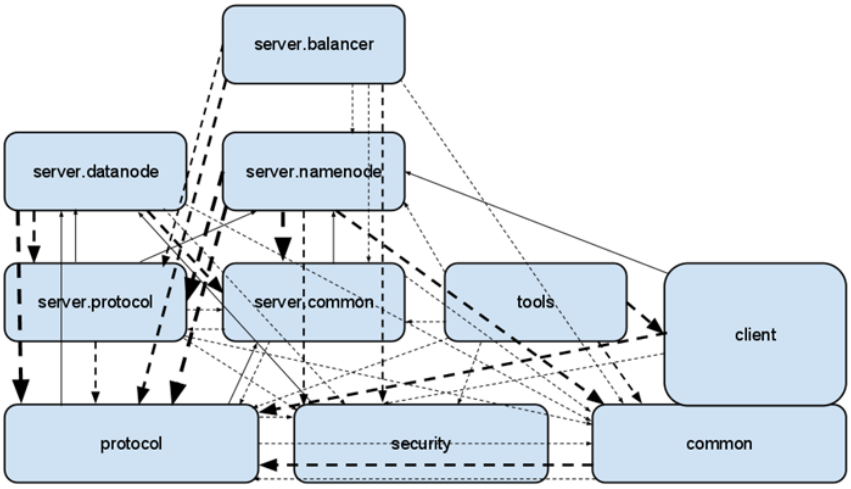


FIGURE A.8: Hadoop Framework: Module View[1]

A.3.2 Hadoop Map-Reduce Architecture

As we already said, Hadoop is a Map-Reduce framework. It is implemented based on a programming paradigm called Map-Reduced. *The Map*: A map transform is provided to transform an input data row of key and value to an output key.value: That is, for an input it returns a list containing zero or more (key,value) pairs:

The Reduce: A reduce transform is provided to take all values for a specific key, and generate a new list of the reduced output.

Hadoop implements this programming paradigm through the following components:

JobTracker: Job Tracker oversees and coordinates the parallel processing of data using Map Reduce. In the Master-Slave architectural style, Job Tracker plays the role of master node.

TaskTracker: Does the actual work given by Job Tracker- It does either map or reduce. In the Master-Slave architectural style, Task Trackers play the role of slave nodes.

[1]Included with permission from Dr. Rick Kazman. Original Source: http://itm-vm.shidler.hawaii.edu/HDFS/

A.3.3 Combined Architectural View

The combined view shows how different components of data layer (HDFS) and paralleling layer (Map-Reduce) work together. Figure A.9 shows different categories of machines and components deployed on them. The Master nodes oversee the two key functional pieces that make up Hadoop: storing lots of data (HDFS), and running parallel computations on all that data (Map Reduce). The Name Node coordinates the data storage function (HDFS), while the Job Tracker coordinates the parallel processing of data using Map Reduce operations. Slave Nodes make up the majority of machines on the cluster and do all the labour work of storing the data and running the computations. Each slave node runs both a Data Node and Task Tracker threads which communicate with and get instructions from their master nodes. The Task Tracker thread is a slave to the Job Tracker, the Data Node thread is a slave to the Name Node.

Client machines must have Hadoop installed, but are neither a Master or a Slave. The primary responsibilities of the Client machine are to load data into the cluster, submit Map Reduce jobs describing how that data should be processed, and then retrieve or view the results of the job when its finished. These roles are logical, it means that in small clusters (40 nodes) a single node (server) can play multiple roles, such as both Job Tracker and Name Node. While in large clusters it is common to have distinct machines for each role.

The summary of cooperations among these nodes is: Client machine submits the Map Reduce job to the Job Tracker, asking ”How many times does Refund occur in File.txt“ The Job Tracker consults the Name Node to learn which Data Nodes have blocks of File.txt. The Job Tracker then provides the Task Tracker running on those nodes with the Java code required to execute the Map computation on their local data.

The Task Tracker starts a Map task and monitors the tasks progress. The Task Tracker provides heartbeats and task status back to the Job Tracker.

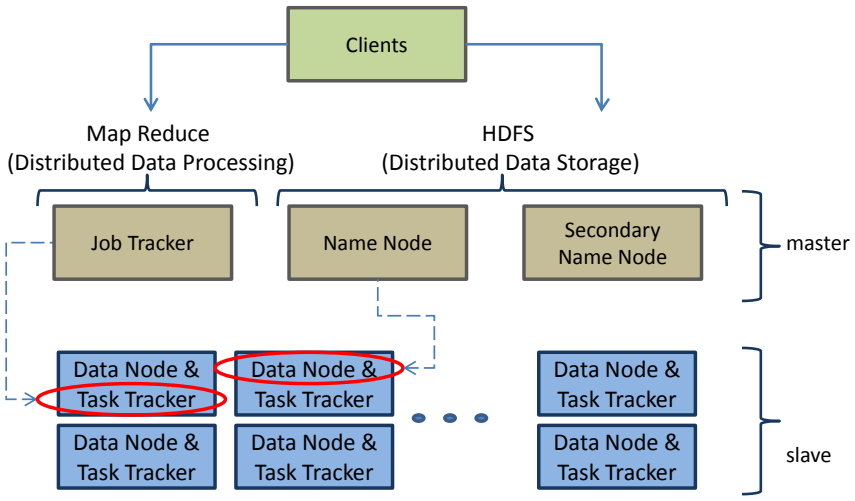


FIGURE A.9: A Combined View: Server Roles in Hadoop

A.3.4 HDFS Architectural Issues

HDFS does not have any interesting layering. Its portability concerns are, by and large, addressed by the technique of "implement in Java". The governing architectural pattern in HDFS is a master-slave style, which is a run-time structure. While modifiability has been important in Hadoop, it has been addressed simply by keeping the code base at a relatively modest size and by having a significant number of committers spending considerable time learning and mastering this code base.

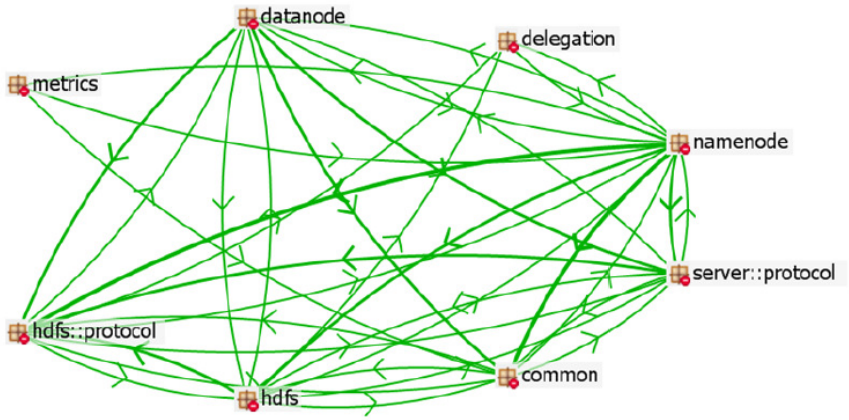


FIGURE A.10: HDFS Reverse Engineered Code Structure[2]

[2]Included with permission from Dr. Rick Kazman. Original Source: [17]

A.3.4.1 Availability

Availability is managed by maintaining multiple replicas of each block in an HDFS file, recognizing failure in a DataNode or corruption of a block, and having mechanisms to replace a failed DataNode or a corrupt block.

Furthermore, the data copy is done through pipeline schema facilitating the speed of file transfer. In this schema as the subsequent blocks of a file are written, the initial node in the pipeline will vary for each block, spreading around the hot spots of in-rack and cross-rack traffic for replication.

Detecting Failure:

The Name Node is the master server coordinating the data node servers in the cluster. Name node contains all the file system metadata for the cluster and and coordinates access to data located on data nodes. Data Nodes send heartbeat messages to the Name Node every 3 seconds via a TCP handshake, using the same port number defined for the Name Node daemon, which is usually TCP 9000. The heartbeat is implemented by piggybacking, therefore every tenth heartbeat is a Block Report, where the Data Node tells the Name Node about all the blocks it has. The block reports are used by the Name Node to build its metadata and insure (3) replicas of the block exist on different nodes, in different racks. In this design, if the Name Node is down the HDFS is down.

Re-replicating the Missing Replicas Missing heartbeat message means the loss of data on a Data Node. Name Node uses the block reports it had been receiving from the dead data node, to decide which blocks of data are lost along with the node and therefore it makes the decision to re-replicate those blocks to other Data Nodes. Name node will consult the Rack Awareness data in order to maintain the two copies in one rack, one copy in another rack replica rule when deciding which Data Node should receive a new copy of the blocks. This rule guarantees that if a rack of servers crashes or fall of the network because of the rack switch failure the client will lose all the data. Figure shows this process.

Single Point of Failure in HDFS Name node could be a single point of failure in HDFS. In the case of an unplanned event such as a machine crash, the cluster would be unavailable until an operator restarted the NameNode, or planned maintenance events such as software or hardware upgrades on the NameNode machine would result in windows of cluster downtime. This means that

the availability of the whole HDFS depends on the health and availability of Name Node. Standby tactic with the following details is used to avoid single point of failure in Hadoop (Figure A.11).

- In a typical HA cluster, two separate machines are configured as NameNodes. At any point in time, exactly one of the NameNodes is in an Active state, and the other is in a Standby state.
- The Active NameNode is responsible for all client operations in the cluster, while the Standby is simply acting as a slave, maintaining enough state to provide a fast failover if necessary.
- To keep the state synchronized, the current implementation requires that the two nodes both have access to a directory on a shared storage device (eg an NFS mount from a NAS)- Point of Failure - This restriction will likely be relaxed in future versions.
- When any namespace modification is performed by the Active node, it durably logs a record of the modification to an edit log file stored in the shared directory. The Standby node is constantly watching this directory for edits, and as it sees the edits, it applies them to its own namespace.
- For fast failover, the Standby node must have up-to-date information regarding the location of blocks in the cluster. Therefore the DataNodes are configured with the location of both NameNodes, and send block location information and heartbeats to both.
- Only manual failover is supported. HA NameNodes are incapable of automatically detecting a failure of the Active NameNode, and instead rely on the operator to manually initiate a failover. Automatic failure detection and initiation of a failover will be implemented in future versions.

A.3.4.2 Security

The major security goal of HDFS is to keep data in HDFS secure from unauthorized access. To achieve this goal, HDFS architecture adopted the following architectural guidelines.

- Users must be authenticated

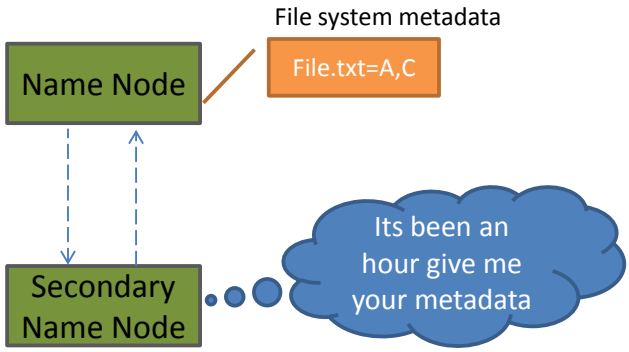


FIGURE A.11: Synchronization between Primary NameNode and Secondary NameNode

- Since Map/Reduce run applications as user, they must authenticate the users.
- Since servers HDFS or Map/Reduce are entrusted with user credentials, they must also be authenticated.
- Kerberose is the key authentication system in HDFS.
- Security on/off option to handle the trade-off between performance and security. Specially in cases which the cluster is used by a single user or company and security is a less concern.

To prevent unauthorized HDFS access, it is decided that all HDFS clients must be authenticated. This includes tasks running as part of MapReduce jobs and Submitted jobs. Users must also authenticate servers, Otherwise fraudulent servers could steal credentials. The idea in the current version of the Hadoop is to integrate Hadoop with Kerberos to provide well tested open source distributed authentication system.

A few major authentication issues of Hadoop were that Hadoop 0.20 completely trusted the user, also user passes their username and groups over wire. The next releases of Hadoop added the feature request to have authentication on both RPC and Web UI. About authorization, HDFS had owners, groups and permissions since 0.16. However Map/Reduce had nothing in version 0.20.

A.3.4.3 Performance

Scheduling is one of the key tactic used to tune the performance on Hadoop. This framework implements a set of scheduling mechanisms **FIFO**: Under Hadoop's default FIFO scheduler as

soon as a job is sent to Hadoop for execution, the JobTracker will assign as many TaskTrackers as necessary to process that job. This strategy has express Checkout for smaller Jobs. Therefore it tends to follow a fair strategy in scheduling the tasks, avoiding starvation.

Fair Scheduler: Multiple jobs, each having a Pool. Each Pool gets a guaranteed number of task slots (map/reduce). "fair sharing" such that each job gets roughly an equal amount of compute resource after exceeding the pool size.

Capacity Scheduler: shares similar goals with the Fair Scheduler. The Capacity Scheduler works on queues rather than pools. Within a queue, jobs have priority. Queues are guaranteed a fraction of the capacity of the grid. Each queue enforces a limit on the percentage of resources allocated to a user at any given time, if there is competition for them

Examining TaskTrackers: Heartbeat tactic and some performance metrics are used to detect task failure, or tasks which do not preform well. Therefore those tasks will be killed and restarted.

Pooling: is used in various parts to increase performance.

Load Balancing: is used to balance data and workload on various computers of the Hadoop cluster.

Nodes in Hadoo cluster are classified asÂăhighly-utilized,Âăaverage-utilized, andÂăunder-utilized. First, it acquires neighborhood details: When the load increases in a DataNode to the threshold level, it sends a request to the NameNode. The NameNode had information about the load levels of the specific DataNode's nearest neighbors. Loads are compared by the NameNode and then the details about the free-est neighbor nodes are sent to the specific DataNode.

Next, the DataNodes go to work: Each DataNode compares its own load amount with the sum of the load amount of its nearest neighbors. If a DataNode's load level is greater than the sum of its neighbors, then load-destination nodes (direct neighbors AND other nodes) will be chosen at random. Load requests are then sent to the destination nodes.

Last, the request is received: Buffers are maintained at every node to received load requests. A message passing interface (MPI) manages this buffer. A main thread will listen to the buffered queue and will service the requests it receives. The nodes enter the loadbalancing execution phase.

Table A.3: Instances of Architectural Tactics in Apache Hadoop

Tactic	Classes	Explanation	Package Name or Subsystem
Heartbeat	27	HDFS uses a master/slave architecture with replication. All slaves send a heartbeat message to the master (server) indicating their health status. Master replicates a failed node (slave).	MapReduce Subsystem
		The MapReduce subsystem uses heartbeat with piggybacking to check the health and execution status of each task running on a cluster.	HDFS Subsystem
Resource Pooling	36	MapReduce uses thread pooling to improve performance of many tasks e.g. to run the map function.	mapred package
	7	A global compressor/decompressor pool used to save and reuse codecs.	compress package
	47	Block pooling is used to improve performance of the distributed file system.	HDFS subsystem
	5	Combines scheduling & job pooling . Organizes jobs into "pools", and shares resources between pools.	MapReduce subsystem
Scheduling	88	Scheduling services are used to execute tasks and jobs. These include fair-, dynamic-, & capacity-scheduling	common & MapReduce
Audit Trail	4	Audit log captures users' activities and authentication events.	mapred package
Authentication	35	Uses Kerberos authentication for direct client access to HDFS subsystems.	security package
		The MapReduce framework uses a DIGEST-MD5 authentication scheme.	MapReduce & HDFS subsys.
Secure Session	35	Uses Kerberos authentication for direct client access to HDFS subsystems.	security package
		The MapReduce framework uses a DIGEST-MD5 authentication scheme.	MapReduce & HDFS subsys.
Asynchronous Communication	13	Handles communication with all the NodeManagers and provides asynchronous updates on getting responses from them	mapreduce & datanode
Hash Based Method Authenti-cation (HMAC)	8	Verifies message replies using base64Hash.	security package & NameNode
		Authorizes queue submissions based on symmetric private key HMAC/SHA1.	MapReduce & dynamic-scheduler.
Role Based Access Control	39	Authorizes access to the directories, files as well as operations on data files	HDFS & fs & NameNode
		authorizes access to the queue of jobs.	MapReduce.
		An authorization manager which handles service-level authorization.	Security.
CheckPoint	32	Periodic checkpoints of the namespace to keep NameNode and Backup NameNode in synch and help Namenode restart.	HDFS

Bibliography

[1] *Apache-Hadoop Design documents.* http://hadoop.apache.org /common/docs/current/hdfs-design.html.

[2] Chromium projects, design documents. *http://www.chromium.org/developers/design-documents*.

[3] U.s. food and drug administration. http://www.fda.gov/.

[4] Ieee standard glossary of software engineering terminology, 1990.

[5] *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[6] Guidelines for the oversight of software change impact analyses used to classify software changes as major or minor. Federal Aviation Administration Notice 8110.85, May 2000.

[7] *Deriving Architectural Tactics: A Step Toward Methodical Architectural Design.* Technical Report, Software Engineering Institute, 2003.

[8] Grand Challenges, Benchmarks, and TraceLab: Developing Infrastructure for the Software Traceability Research Community. *International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE) 6* (2011).

[9] Abadi, A., Nisenson, M., and Simionovici, Y. A traceability technique for specifications. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on* (June 2008), pp. 103–112.

[10] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng. 28*, 10 (Oct. 2002), 970–983.

[11] Aplin, J. Primary flight computers for the boeing 777. *Microprocessors and Microsystems 20*, 8 (1997), 473–478.

[12] Babar, M. A., Dingsyr, T., Lago, P., and van Vliet, H. *Software Architecture Knowledge Management: Theory and Practice*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[13] Babar, M. A., and Gorton, I. A tool for managing software architecture knowledge. In *Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent* (Washington, DC, USA, 2007), SHARK-ADI '07, IEEE Computer Society, pp. 11–.

[14] Bachmann, F., Bass, L., and Klein, M. *Deriving Architectural Tactics: A Step Toward Methodical Architectural Design*. Technical Report, Software Engineering Institute, 2003.

[15] Bachmann, P. Deferred cancellation: a behavioral pattern. In *Proceedings of the 15th Conference on Pattern Languages of Programs* (New York, NY, USA, 2008), PLoP '08, ACM, pp. 18:1–18:17.

[16] Baeza-Yates, R. A., and Ribeiro-Neto, B. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[17] Bass, L., Kazman, R., and Ozkaya, I. Developing architectural documentation for the hadoop distributed file system. In *Open Source Systems: Grounding Research*, S. Hissam, B. Russo, M. MendonÃğa Neto, and F. Kon, Eds., vol. 365 of *IFIP Advances in Information and Communication Technology*. Springer Berlin Heidelberg, 2011, pp. 50–61.

[18] Booch, G. Draw me a picture. *IEEE Software 28* (2011), 6–7.

[19] Bosch, J. *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[20] Breiman, L. Bagging predictors. *Mach. Learn. 24*, 2 (Aug. 1996), 123–140.

[21] Breiman, L. Bagging predictors. *Mach. Learn. 24*, 2 (Aug. 1996), 123–140.

[22] Brown, W. J., Malveau, R. C., McCormick, III, H. W., and Mowbray, T. J. *AntiPatterns: refactoring software, architectures, and projects in crisis.* John Wiley & Sons, Inc., New York, NY, USA, 1998.

[23] Burge, J. E., and Brown, D. C. Software engineering using rationale. *Journal of Systems and Software 81*, 3 (2008), 395–413.

[24] C4ISR Architecture Working Group. C4ISR Architecture Framework, Version 2.0. Washington, D. D. o. D. . *http://www.afcea.org/education/courses/archfwk2.pdf.*

[25] Capilla, R., Nava, F., Pérez, S., and Dueñas, J. C. A web-based tool for managing architectural design decisions. *SIGSOFT Softw. Eng. Notes 31* (Sept. 2006).

[26] Castro, J., Kolp, M., and Mylopoulos, J. Towards requirements-driven information systems engineering: the tropos project. *Inf. Syst. 27*, 6 (Sept. 2002), 365–389.

[27] Cleland-Huang, J., Berenbach, B., Clark, S., Settimi, R., and Romanova, E. Best practices for automated traceability. *Computer 40*, 6 (2007), 27–35.

[28] Cleland-Huang, J., Chang, C. K., and Christensen, M. J. Event-based traceability for managing evolutionary change. *IEEE Trans. Software Eng. 29*, 9 (2003), 796–810.

[29] Cleland-Huang, J., Chang, C. K., and Ge, Y. Supporting event based traceability through high-level recognition of change events. In *COMPSAC* (2002), pp. 595–602.

[30] Cleland-Huang, J., Chang, C. K., and Wise, J. C. Automating performance-related impact analysis through event based traceability. *Requir. Eng. 8*, 3 (2003), 171–182.

[31] Cleland-Huang, J., Czauderna, A., Gibiec, M., and Emenecker, J. A machine learning approach for tracing regulatory codes to product specific requirements. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on* (May 2010), vol. 1, pp. 155–164.

[32] Cleland-Huang, J., Czauderna, A., Gibiec, M., and Emenecker, J. A machine learning approach for tracing regulatory codes to product specific requirements. In *ICSE (1)* (2010), pp. 155–164.

[33] Cleland-Huang, J., Gotel, O., Huffman Hayes, J., Mader, P., and Zisman, A. Software traceability: Trends and future directions. In *Proc. of the 36th International Conference on Software Engineering (ICSE), Hyderabad, India* (2014).

[34] Cleland-Huang, J., Marrero, W., and Berenbach, B. Goal-centric traceability: Using virtual plumblines to maintain critical systemic qualities. *IEEE Trans. Software Eng. 34*, 5 (2008), 685–699.

[35] Cleland-Huang, J., and Schmelzer, D. Dynamically tracing non-functional requirements through design pattern invariants, 2003.

[36] Cleland-Huang, J., Settimi, R., Duan, C., and Zou, X. Utilizing supporting evidence to improve dynamic requirements traceability. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering* (Washington, DC, USA, 2005), RE '05, IEEE Computer Society, pp. 135–144.

[37] Cleland-Huang, J., Settimi, R., Zou, X., and Solc, P. Automated detection and classification of non-functional requirements. *Requir. Eng. 12*, 2 (2007), 103–120.

[38] Cohen, W., and Singer, Y. A simple and fast and and effective rule learner. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence* (1999), pp. 335–342.

[39] Cotroneo, D., Natella, R., Pietrantuono, R., and Russo, S. A survey of software aging and rejuvenation studies. *J. Emerg. Technol. Comput. Syst. 10*, 1 (Jan. 2014), 8:1–8:34.

[40] Cuddeback, D., Dekhtyar, A., and Hayes, J. H. Automated requirements traceability: The study of human analysts. In *RE* (2010), pp. 231–240.

[41] Cuddeback, D., Dekhtyar, A., Hayes, J. H., Holden, J., and Kong, W.-K. Towards overcoming human analyst fallibility in the requirements tracing process: Nier track. In *ICSE* (2011), pp. 860–863.

[42] Cysneiros, L. M., and Yu, E. Non-functional requirements: A comprehensive approach.

[43] Davis, A. M. *Software requirements: objects, functions, and states.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[44] De Lucia, A., Oliveto, R., and Sgueglia, P. Incremental approach and user feedbacks: a silver bullet for traceability recovery. In *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on* (Sept 2006), pp. 299–309.

[45] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. Indexing by latent semantic analysis. *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE 41*, 6 (1990), 391–407.

[46] Egyed, A., Biffl, S., Heindl, M., and Grünbacher, P. Determining the cost-quality trade-off for automated software traceability. In *ASE* (2005), pp. 360–363.

[47] Eramo, R., Cortellessa, V., Pierantonio, A., and Tucci, M. Performance-driven architectural refactoring through bidirectional model transformations. In *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures* (New York, NY, USA, 2012), QoSA '12, ACM, pp. 55–60.

[48] Fairbanks, G. *Just Enough Software Architecture A Risk-Driven Approach.* Marshall & Brainerd, 2010.

[49] for Advancing Software-Intensive Systems Producibility; National Research Council, C. *Critical Code: Software Producibility for Defense.* The National Academies Press, 2010.

[50] Freund, Y., and Schapire, R. E. Experiments with a new boosting algorithm. In *Thirteenth International Conference on Machine Learning* (San Francisco, 1996), Morgan Kaufmann, pp. 148–156.

[51] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[52] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[53] Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. Identifying architectural bad smells. In *Proceedings of the 2009 European Conference on Software Maintenance and*

Reengineering (Washington, DC, USA, 2009), CSMR '09, IEEE Computer Society, pp. 255–258.

[54] Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. Toward a catalogue of architectural bad smells. In *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems* (Berlin, Heidelberg, 2009), QoSA '09, Springer-Verlag, pp. 146–162.

[55] Garlan, D., and Shaw, M. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering* (1993), Publishing Company, pp. 1–39.

[56] Genkin, A., Lewis, D. D., and Madigan, D. Large-scale bayesian logistic regression for text categorization. *Technometrics 49* (August 2007), 291–304(14).

[57] Gotel, O., Cleland-Huang, J., Hayes, J., Zisman, A., Egyed, A., GrÃijnbacher, P., Dekhtyar, A., Antoniol, G., Maletic, J., and MÃĭder, P. Traceability fundamentals. In *Software and Systems Traceability*, J. Cleland-Huang, O. Gotel, and A. Zisman, Eds. Springer London, 2012, pp. 3–22.

[58] Gotel, O., Cleland-Huang, J., Hayes, J. H., Zisman, A., Egyed, A., GrÃijnbacher, P., Dekhtyar, A., Antoniol, G., Maletic, J., and MÃĭder, P. *Traceability Fundamentals*. Springer-Verlag London Limited, 2012, pp. 3–22.

[59] Gross, D., and Yu, E. From non-functional requirements to design through patterns. *Requirements Engineering 6* (2000), 18–36.

[60] Group, I. A. W. Ieee std 1471-2000, recommended practice for architectural description of software-intensive systems. Tech. rep., IEEE, 2000.

[61] Grundy, J. Aspect-oriented requirements engineering for component-based software systems. *2012 20th IEEE International Requirements Engineering Conference (RE) 0* (1999), 84.

[62] Guide, R. T. P. *Online at: http://www2.cdc.gov/cdcup/library/practices*.

[63] Guo, G. Y., Atlee, J. M., and Kazman, R. A software architecture reconstruction method. In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)* (Deventer, The Netherlands, The Netherlands, 1999), WICSA1, Kluwer, B.V., pp. 15–34.

[64] Hanmer, R. *Patterns for Fault Tolerant Software*. Wiley Series in Software Design Patterns, 2007.

[65] Hart, J., King, E., Miotto, P., and Lim, S. *Orion GN&C Architecture for Increased Spacecraft Automation and Autonomy Capabilities*, August 2008.

[66] Hayes, J., Dekhtyar, A., and Sundaram, S. Advancing candidate link generation for requirements tracing: the study of methods. *Software Engineering, IEEE Transactions on 32*, 1 (Jan 2006), 4–19.

[67] HegedÅśs, P., BÃąn, D., Ferenc, R., and GyimÃŞthy, T. Myth or reality? analyzing the effect of design patterns on software maintainability. In *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity*, T.-h. Kim, C. Ramos, H.-k. Kim, A. Kiumi, S. Mohammed, and D. ÅŽlÄŹzak, Eds., vol. 340 of *Communications in Computer and Information Science*. Springer Berlin Heidelberg, 2012, pp. 138–145.

[68] Hofmeister, C., Nord, R., and Soni, D. *Applied software architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[69] Huang, G., Mei, H., and Yang, F.-Q. Runtime recovery and manipulation of software architecture of component-based systems. *Automated Software Engg. 13*, 2 (Apr. 2006), 257–281.

[70] (IRG), N. A. I. R. G. *Online at: http://ti.arc.nasa.gov/tech/asr/intelligent-robotics/*, 2012.

[71] Izurieta, C., and Bieman, J. M. How software designs decay: A pilot study of pattern evolution. In *ESEM* (2007), pp. 449–451.

[72] Jansen, A., and Bosch, J. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 109–120.

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

[74] Joachims, T. Text categorization with suport vector machines: Learning with many relevant features. In *Proceedings of the 10th European Conference on Machine Learning* (London, UK, UK, 1998), ECML '98, Springer-Verlag, pp. 137–142.

[75] Kazman, R., Barbacci, M., Klein, M., Carrière, S. J., and Woods, S. G. Experience with performing architecture tradeoff analysis. In *Proceedings of the 21st international conference on Software engineering* (New York, NY, USA, 1999), ICSE '99, ACM, pp. 54–63.

[76] Kazman, R., Klein, M., and Clements, P. Atam: A method for architecture evaluation. *Software Engineering Institute* (2000).

[77] Kruchten, P. The 4+1 view model of architecture. *IEEE Softw. 12* (November 1995), 42–50.

[78] Kruchten, P. An ontology of architectural design decisions. *Groningen Workshop on Software Variability management* (2004), 55–62.

[79] Kruchten, P., Capilla, R., and Dueas, J. C. The decision view's role in software architecture practice. *IEEE Software 26*, 2 (2009), 36–42.

[80] L. Chung, B. Nixon, E. Y., and Mylopoulos, J. Non-functional requirements in software engineering., 2000.

[81] Len Bass, Paul Clements, R. K. *Software Architecture in Practice*. 2000.

[82] Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., and Watkins, C. Text classification using string kernels. *J. Mach. Learn. Res. 2* (Mar. 2002), 419–444.

[83] Lung, C.-H., Zaman, M., and Nandi, A. Applications of clustering techniques to software partitioning, recovery and restructuring. *J. Syst. Softw. 73*, 2 (Oct. 2004), 227–244.

[84] Mancoridis, S., Mitchell, B. S., Chen, Y., and Gansner, E. R. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the IEEE International Conference on Software Maintenance* (Washington, DC, USA, 1999), ICSM '99, IEEE Computer Society, pp. 50–.

[85] Marcus, A., and Maletic, J. I. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering* (Washington, DC, USA, 2003), ICSE '03, IEEE Computer Society, pp. 125–135.

[86] Medvidovic, N., and Egyed, A. Stemming architectural erosion by coupling architectural discovery and recovery, 2003.

[87] Merkle, B. Stop the software architecture erosion. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion* (New York, NY, USA, 2010), SPLASH '10, ACM, pp. 295–297.

[88] Mirakhorli, M., and Cleland Huang, J. A decision-centric approach for tracing reliability concerns in embedded software systems. In *Proceedings of the Workshop on Embedded Software Reliability (ESR), held at ISSRE10* (November 2010).

[89] Mirakhorli, M., and Cleland-Huang, J. Tracing architectural concerns in high assurance systems: (nier track). In *ICSE* (2011), pp. 908–911.

[90] Mirakhorli, M., and Cleland-Huang, J. Transforming trace information in architectural documents into re-usable and effective traceability links. In *Proceedings of the 6th workshop on Sharing Architectural Knowledge* (May 2011).

[91] Mirakhorli, M., and Cleland-Huang, J. Using tactic traceability information models to reduce the risk of architectural degradation during system maintenance. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance* (Washington, DC, USA, 2011), ICSM '11, IEEE Computer Society, pp. 123–132.

[92] Mirakhorli, M., Shin, Y., Cleland-Huang, J., and Cinar, M. A tactic centric approach for automating traceability of quality concerns. In *International Conference on Software Engineering, ICSE (1)* (2012).

[93] Mirakhorli, M., Shin, Y., Cleland-Huang, J., and Cinar, M. A tactic-centric approach for automating traceability of quality concerns. In *Proceedings of the 2012 International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), ICSE 2012, IEEE Press, pp. 639–649.

[94] Murphy, G. C., Notkin, D., and Sullivan, K. J. Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Softw. Eng. 27*, 4 (Apr. 2001), 364–380.

[95] NASA's, and Robots. *Online at: http://prime.jsc.nasa.gov/ ROV/nlinks.html*, 2008.

[96] Ormandjieva, M. K. O. Towards an aspectoriented software development model with tractability mechanism, 2006.

[97] Parnas, D. Designing software for ease of extension and contraction. *Software Engineering, IEEE Transactions on SE-5*, 2 (March), 128–138.

[98] Parnas, D. L. Designing software for ease of extension and contraction. In *Proceedings of the 3rd international conference on Software engineering* (Piscataway, NJ, USA, 1978), ICSE '78, IEEE Press, pp. 264–277.

[99] Parnas, D. L. Software aging. In *Proceedings of the 16th international conference on Software engineering* (Los Alamitos, CA, USA, 1994), ICSE '94, IEEE Computer Society Press, pp. 279–287.

[100] Perry, D. E., and Wolf, A. L. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes 17* (October 1992), 40–52.

[101] Poshyvanyk, D., Gethers, M., and Marcus, A. Concept location using formal concept analysis and information retrieval. *ACM Trans. Softw. Eng. Methodol. 21*, 4 (Feb. 2013), 23:1–23:34.

[102] Quinlan, J. R. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[103] Ramesh, B., and Edwards, M. Issues in the development of a requirements traceability model. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on* (1993), pp. 256–259.

[104] Ramesh, B., and Jarke, M. Toward reference models for requirements traceability. *IEEE Trans. Softw. Eng. 27* (January 2001), 58–93.

[105] Rashid, A., Moreira, A., and Araújo, J. Modularisation and composition of aspectual requirements. In *Proceedings of the 2nd international conference on Aspect-oriented software development* (New York, NY, USA, 2003), AOSD '03, ACM, pp. 11–20.

[106] Rashid, A., Sawyer, P., Moreira, A., and Araujo, J. Early aspects: a model for aspect-oriented requirements engineering. In *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on* (2002), pp. 199 – 202.

[107] Rasool, G., and Mäder, P. Flexible design pattern detection based on feature types. In *ASE* (2011), pp. 243–252.

[108] Rosik, J., Le Gear, A., Buckley, J., and Ali Babar, M. An industrial case study of architecture conformance. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement* (New York, NY, USA, 2008), ESEM '08, ACM, pp. 80–89.

[109] Rumbaugh, J., Jacobson, I., and Booch, G., Eds. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[110] Salazar-zÃąrate, G., and Botella, P. Use of uml for modeling non-functional aspects, 2000.

[111] Sartipi, K. Software architecture recovery based on pattern matching. In *Proceedings of the International Conference on Software Maintenance* (Washington, DC, USA, 2003), ICSM '03, IEEE Computer Society, pp. 293–.

[112] (SEI), C. M. S. E. I. *Online at: http://www.sei.cmu.edu/*.

[113] Sieriorek, D., and Narasimhan, P. Fault tolerant architectures for space and avionics applications. *NASA Ames Research* (http://ic.arc.nasa.gov/projects/ishem/Papers/Siewi).

[114] Siewiorek, D. P., and Narasimhan, P. Fault-tolerant architectures for space and avionics applications. NASA Ames Research http://ic.arc.nasa.gov/projects/ishem/Papers/Siewi.

[115] Tamblyn, S., Hinkel, H., and Saley, D. *NASA Exploration Systems Architecture Study (ESAS) Final Report*, 2005.

[116] Tamblyn, S., Hinkel, H., and Saley, D. *Crew Exploration Vehicle (CEV) Reference Guidance, Navigation, and Control (GN&C) Architecture*, February 2007.

[117] Tang, A., Avgeriou, P., Jansen, A., Capilla, R., and Babar, M. A. A comparative study of architecture knowledge management tools. *Journal of Systems and Software 83*, 3 (2010), 352 – 370.

[118] Tang, A., Jin, Y., and Han, J. A rationale-based architecture model for design traceability and reasoning. *Journal of Systems and Software 80*, 6 (2007), 918 – 934.

[119] Taylor, R., Medvidovic, N., and Dashofy, E. *Software Architecture: Foundations, Theory, and Practice*. John Wiley and Sons, 2009.

[120] Tekinerdoğan, B., Hofmann, C., Akşit, M., and Bakker, J. Metamodel for tracing concerns across the life cycle. In *Proceedings of the 10th international conference on Early aspects: current challenges and future directions* (Berlin, Heidelberg, 2007), Springer-Verlag, pp. 175–194.

[121] Tong, S., and Koller, D. Support vector machine active learning with applications to text classification. *J. Mach. Learn. Res. 2* (Mar. 2002), 45–66.

[122] Tracelab. *Online at: http://www.coest.org*, 2013.

[123] Tyree, J., and Akerman, A. Architecture decisions: Demystifying architecture. *IEEE Softw. 22*, 2 (Mar. 2005), 19–27.

[124] University of California, I. The sourcerer project. sourcerer.ics.uci.edu.

[125] van Gurp, J., and Bosch, J. Design erosion: problems and causes. *J. Syst. Softw. 61*, 2 (Mar. 2002), 105–119.

[126] van Gurp, J., Bosch, J., and Svahnberg, M. On the notion of variability in software product lines. In *WICSA* (2001), pp. 45–54.

[127] van Gurp, J., Brinkkemper, S., and Bosch, J. Design preservation over subsequent releases of a software product: a case study of baan erp: Practice articles. *J. Softw. Maint. Evol. 17* (July 2005), 277–306.

[128] van Lamsweerde, A., and Letier, E. From object orientation to goal orientation: A paradigm shift for requirements engineering. In *RISSEF* (2002), M. Wirsing, A. Knapp, and S. Balsamo, Eds., vol. 2941 of *Lecture Notes in Computer Science*, Springer, pp. 325–340.

[129] Vokáč, M., Tichy, W., Sjøberg, D. I. K., Arisholm, E., and Aldrin, M. A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment. *Empirical Softw. Engg. 9*, 3 (Sept. 2004), 149–195.

[130] Warmer, J., and Kleppe, A. *The Object Constraint Language: Getting Your Models Ready for MDA*, 2 ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[131] Williams, B. J., and Carver, J. C. Characterizing software architecture changes: A systematic review. *Information & Software Technology 52*, 1 (2010), 31–51.

[132] Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R., and Wood, B. Attribute-driven design (add), version 2.0, 2006-023.

[133] Yadla, S., Hayes, J. H., and Dekhtyar, A. Tracing requirements to defect reports: an application of information retrieval techniques. *Innovations in Systems and Software Engineering 1*, 2 (2005), 116–124.

[134] Yu, E. Towards modelling and reasoning support for early-phase requirements engineering. In *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on* (jan 1997), pp. 226 –235.

[135] Zhu, L., and Gorton, I. Uml profiles for design decisions and non-functional requirements. In *Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent* (Washington, DC, USA, 2007), SHARK-ADI '07, IEEE Computer Society, pp. 8–.