A Tactic-Centric Approach for Automating Traceability of Quality Concerns

Mehdi Mirakhorli, Yonghee Shin, Jane Cleland-Huang, and Murat Cinar
School of Computing
DePaul University, Chicago, IL 60604
m.mirakhorli@acm.org, yshin@cdm.depaul.edu, jhuang@cs.depaul.edu, murat.cinar87@gmail.com

***Abstract*—The software architectures of business, mission, or safety critical systems must be carefully designed to balance an exacting set of quality concerns describing characteristics such as security, reliability, and performance. Unfortunately, software architectures tend to degrade over time as maintainers modify the system without understanding the underlying architectural decisions. Although this problem can be mitigated by manually tracing architectural decisions into the code, the cost and effort required to do this can be prohibitively expensive. In this paper we therefore present a novel approach for automating the construction of traceability links for architectural tactics. Our approach 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 Information Model. We train our trace algorithm using code extracted from fifteen performance-centric and safety-critical open source software systems and then evaluate it against the Apache Hadoop framework. Our results show that automatically generated traceability links can support software maintenance activities while helping to preserve architectural qualities.**

***Keywords*-Architecture; traceability; tactics; traceability information models**

I. Introduction

Software traceability provides critical support for a broad range of software engineering activities including impact analysis, regression testing, and compliance verification. Unfortunately, the cost and effort required to establish and maintain effective and accurate traceability links can be inhibitively expensive [6], [16], often resulting in organizations implementing only the minimal traceability needed to comply to regulatory or process guidelines.

This problem is especially evident when it comes to tracing quality concerns, which describe system level attributes such as security, reliability, performance, and safety. Such concerns are often addressed in the solution domain through the strategic adoption of architectural tactics [5], [20], [23], which we informally define as re-usable solutions for satisfying a quality concern. A more formal definition is provided by Bachman et al. who 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" [4]. There are many different kinds of tactics. For example, a system with high reliability requirements might implement the *heartbeat* tactic [5] to monitor availability of a critical component, or the *voting* tactic [5] to increase fault tolerance through integrating and processing information from a set of redundant components. Unfortunately, unless these architectural decisions are fully documented, architectural knowledge can be lost over time [12], [30], and as a result there is a tendency for system quality to degrade as developers modify components without fully understanding the underlying design decisions.

Current software engineering tools, practices, and supporting traceability techniques contribute to this problem through failing to make underlying design decisions and their related quality concerns visible to software engineers [7]. At the same time quality concerns tend to have a cross-cutting impact on the solution, and can therefore affect numerous components and exhibit complex interdependencies, [20], [27]. This introduces a dilemma. On one hand it can be difficult and costly to trace quality concerns into the architectural design, as the end result 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.

This paper therefore presents a novel and cost-effective approach for tracing architecturally significant concerns, specifically those concerns which are implemented through the use of common architectural tactics. The proposed process involves the steps depicted in Figure 1. First, a tactic-classifier identifies all classes related to a given tactic, and then establishes tactic-level traceability through mapping those classes to the relevant tactic. Second, a more finely-tuned classifier is used in conjunction with lightweight structural analysis to identify the subset of classes which play clearly defined roles in the tactic. 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 Information Models (tTIMs) which anchor the traceability process and connect the classified classes to a relevant set of design rationales, requirements, and other related artifacts [26], [28].

The novel contribution of this work is twofold. First, it introduces and evaluates a set of algorithms and processes designed to automatically reconstruct traceability links for architectural tactics. This builds on our prior work [26]–[28],

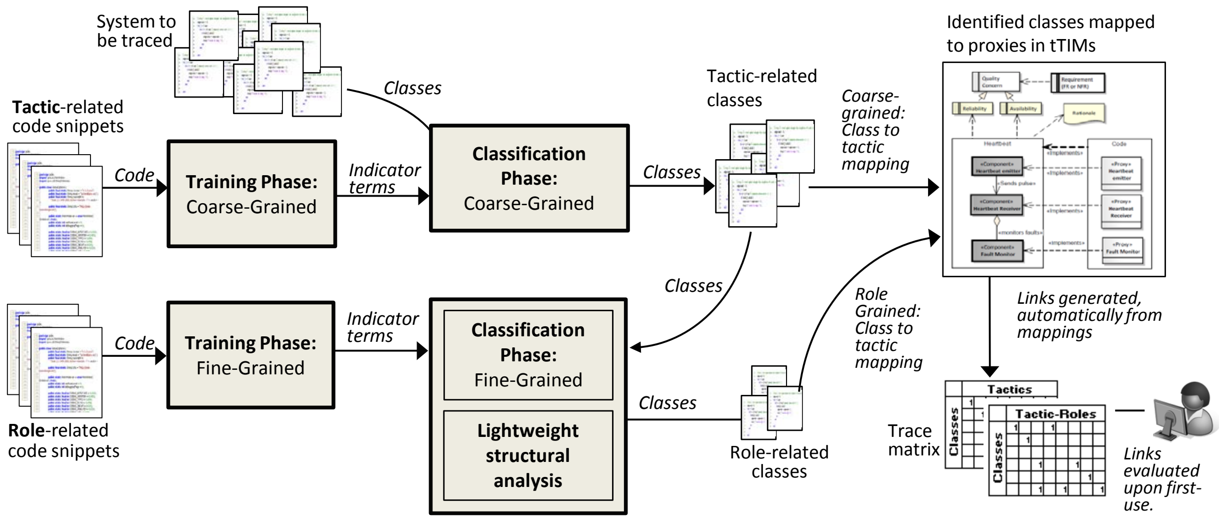

System to be traced
Tactic-related code snippets
Code
Training Phase: Coarse-Grained
Indicator terms
Classes
Classification Phase: Coarse-Grained
Classes
Tactic-related classes
Coarse-grained: Class to tactic mapping
Identified classes mapped to proxies in tTIMs
Code
Training Phase: Fine-Grained
Indicator terms
Classification Phase: Fine-Grained
Lightweight structural analysis
Classes
Classes
Role-related classes
Role Grained: Class to tactic mapping
Role-related code snippets
Links generated, automatically from mappings
Trace matrix
Tactics
Tactic-Roles
Classes
Links evaluated upon first-use.

Figure 1. An Overview of the Tactic-Related Trace Reconstruction Process

which required all traces to be created manually. Second, we contextualize our work through an extended study showing the use of architectural tactics in performance-centric, fault-tolerant software systems.

The remainder of this paper is laid out as follows. Section II provides a more detailed explanation of architectural tactics, and introduces the concept of tactic Traceability Information Models. Sections III, IV, and V describe our tactic classifier and the approach we took to identify tactic related classes and their associated tactic roles. Section VI presents a case study in which traceability links were automatically reconstructed for tactics in the Apache Hadoop framework and then used to support a simulated software maintenance problem. The paper concludes with a discussion of threats to validity, related work, and conclusions.

II. Architectural Tactics

Architectural tactics come in many different shapes and sizes and describe solutions for a wide range of quality concerns [5]. For example, reliability tactics provide solutions for fault mitigation, detection, 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 [18]. Table I depicts the architectural tactics we identified through inspecting the code and supporting documentation of 15 performance-centric, fault-tolerant, open-source software systems. Our analysis focused on 16 different tactics and clearly highlighted the pervasive nature of architectural tactics in the examined systems.

A. Tactics Selected for Study

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 in this paper 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 tactics are defined as follows [5]:
Heartbeat: One 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. This tactic is used to achieve reliability goals.
Scheduling: Resource contentions are managed through scheduling policies such as FIFO (First in first out), fixed-priority, and dynamic priority scheduling.
Resource pooling: Limited resources are shared between clients that do not need exclusive and continual access to a resource. Pooling is typically used for sharing threads, database connections, sockets, and other such resources.
Authentication: Ensures that a user or a remote system is who it claims to be. Authentication is often achieved through passwords, digital certificates, or biometric scans.
Audit trail: A copy of each transaction and associated identifying information is maintained. This audit information can be used to recreate the actions of an attacker, and to support functions such as system recovery and nonrepudiation.

B. Tactic Traceability Information Models

Our approach to tracing architectural tactics builds upon the fundamental concept of tTIMs described in our earlier work [28]. The tTIM concept emerged as a result of an earlier study of tactical architectural decisions which we conducted across a wide range of software intensive systems including the Airbus A320/330/340 family, Boeing 777, Boeing 7J7, NASA robots, and also performance centric systems such as Google Chromium OS [24]. Each tTIM

Table I
An Analysis of Tactics Across Several Open-Source Projects

	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.			●	●						○						

Legend: ● = Included in the code-snippet dataset and used for experiments described in this paper, ○ = For information purposes only

describes the elements needed to trace an individual architectural tactic back to its contribution structures (i.e. quality goals, rationales, intents), and forward to the elements that realize the tactic in both the design and the code. The tTIM includes a set of *roles* describing the essence of the tactic, a set of *semantically typed links* that define relationships between pairs of artifacts, and a set of *trace proxies* which provide mapping points for establishing traceability links.

The tactic itself is modeled as a set of interrelated roles. For example, the *heartbeat* tactic, which is depicted in Figure 2, includes the primary roles of *receiver*, *emitter*, and *fault monitor*. Additional roles, not shown in this figure but described in our earlier work [28], include parameters such as the *heart beat rate* and the *heartbeat checking interval*. In this paper we focus only on the primary roles of each tactic. In addition to roles, the tTIM also includes a set of reusable, semantically-typed traceability links. These include internal links such as ≪*sends pulse*≫ which define relationships between roles in the tactic, as well as a set of external links which are used to establish traceability to the code or the design. Finally, the tTIM contains a set of *trace proxies* which are used to transform the traceability task to a simple mapping task. A developer or analyst has to simply map the proxy onto one or more elements in the architecture and/or code in order to establish traceability. Once these mappings are accomplished, all of the traceability information embedded in the tTIM is automatically inherited by the project. As a result, tTIMs have been shown to reduce the cost and effort of traceability [28].

Traceability can be established at different levels of granularity [13]. Coarse-grained traceability links take less effort to construct and maintain but provide less accurate

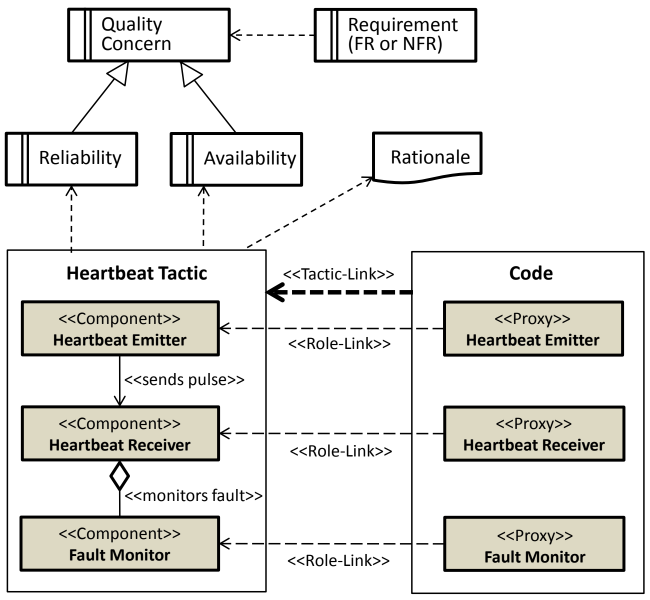


Figure 2. Tactic Traceability Information Model for Heartbeat Tactic

information than finer-grained links when actually used [11]. Our approach therefore supports traceability at either the tactic level or the role-level, as depicted in Figure 2.

III. Identifying Tactic-Related Classes

Although the tactic-detection problem may initially appear to be a special case of design pattern recognition, it turns out to be more challenging. Unlike design patterns which tend to be described in terms of classes and their associations [15], tactics are described in terms of roles and interactions [5]. This means that a single tactic might be implemented using a variety of different design patterns or proprietary designs.

For example we observed the *heartbeat* tactic implemented using (i) direct communication between the emitter and receiver roles *(found in Chat3 and Smartfrog systems)*, (ii) the observer pattern [15] in which the receiver registered as a listener to the emitter *found in the Amalgam system*, (iii) the decorator pattern [15] 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.

As a tactic is not dependent upon a specific structural format, we cannot use structural analysis as the primary means of identification. Our approach therefore 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.

To classify classes according to various tactics we utilized an algorithm that we had previously developed to detect non-functional requirements (NFRs) [10] and to trace regulatory codes [9]. Prior studies demonstrated that this algorithm matched or outperformed standard classification techniques including the naive bayes classifier, standard decision tree algorithm (J48), feature subset selection (FSS), correlation-based feature subset selection (CFS), and various combinations of the above for the specific task of classifying NFRs in the studied datasets [19].

The classifier includes three phases of preparation, training, and classifying which are defined as follows:

A. Preparation

All data is preprocessed using standard information retrieval techniques, and each class and/or tactic description is transformed into a vector of terms.

B. Training

The training phase takes a set of preclassified code segments as input, and produces a set of 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.

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:

Table II
INDICATOR TERMS LEARNED DURING TRAINING

Tactic Name	**Document trained indicator terms**	**Code trained indicator terms**
Heartbeat	heartbeat, fault, detect, messag, period, watchdog, send, tactic, failur, aliv	heartbeat, ping, beat, heart, hb, outbound, puls, hsr, period, isonlin
Scheduling	prioriti, schedul, assign, process, time, queue, robin higher, weight, dispatch	schedul, task, prioriti, prcb, sched, thread, , rtp, weight, tsi
Authentication	authent, password, kerbero, sasl, ident, biometr, verifi, prove, ticket, purport	authent, credenti, challeng, kerbero, auth, login, otp, cred, share, sasl
Resource Pooling	thread, pool, number, worker, task, queue, executor, creat, overhead, min	pool, thread, connect, sparrow, nbp, processor, worker, timewait, jdbc, ti
Audit Trail	audit, trail, record, activ, log, databas, access, action, monitor, user	audit, trail, wizard, pwriter, lthread, log, string, categori, pstmt, pmr

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

C. Classification

During the classification phase, the indicator terms computed in Equation 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:

$$Pr_q(c) = \frac{\sum_{t \in c \cap I_q} Pr_q(t)}{\sum_{t \in I_q} Pr_q(t)} \quad (2)$$

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.

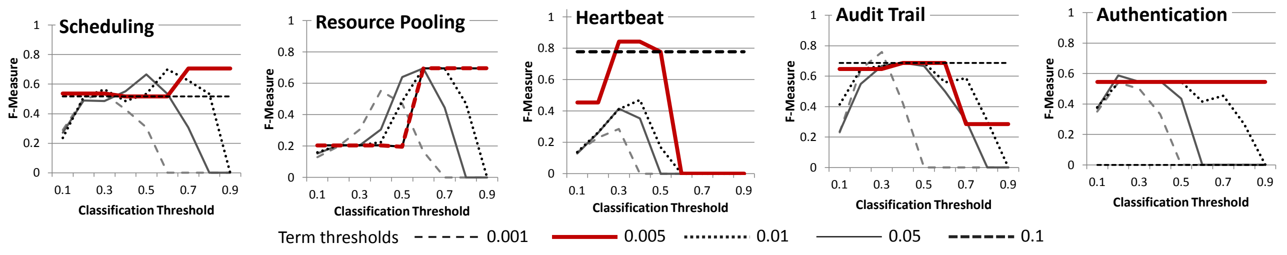
Classes are considered to be related to a given tactic q if the classification score is higher than a selected threshold.

IV. Tactic Level Link Reconstruction

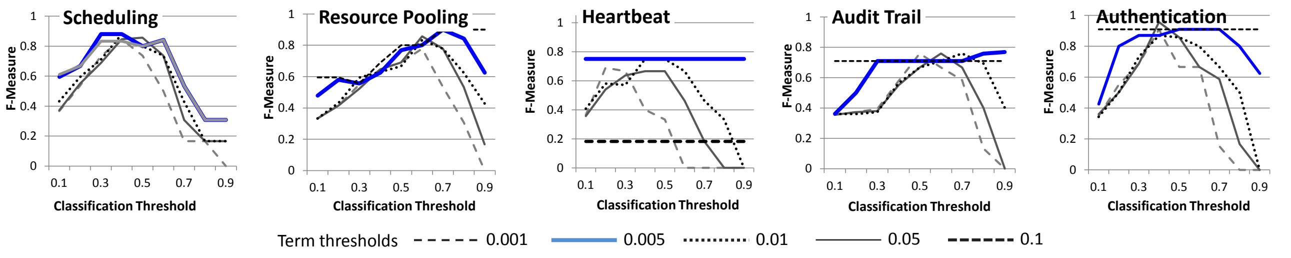
The first step of reconstructing tactic-related traceability links utilizes the classifier described in Equations 1 and 2 to detect classes that implement the targeted tactic. For experimental purposes we investigated two different training methods. The first method trained the classifier using textual descriptions of each tactic, while the second method trained the classifier using code snippets taken from classes implementing each of the tactics.

A. Datasets

Two datasets were developed to support the task of training and evaluating the tactic-grained classifier.



(a) Trained using Tactic Descriptions



(b) Trained using Code Snippets from Tactics implemented in Open Source Systems

Figure 3. Results for Coarse-Grained Detection of Tactic-related Classes at various Classification and Term Thresholds for five Different Tactics

*1) **Tactic Descriptions Dataset**:* For each of the five targeted tactics i.e. *heartbeat*, *resource pooling*, *scheduling*, *audit trail*, and *authentication*, we retrieved ten descriptions of tactics taken from text books, online descriptions, and publications. For training purposes, the dataset also included 20 descriptions of non-tactic-related IT documents. The following text provides an excerpt from a tactic 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.....

*2) **Code Snippets Dataset**:* For each of the five targeted tactics we identified 10 different open-source projects, or parts of a large project, in which the tactic was implemented. For each of these projects we retrieved code segments that were closely related to the tactic. We also retrieved four additional non tactic-related classes for training and testing purposes. The following code represents two methods extracted from a code snippet for the audit tactic.

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

B. Experiments

Experiments were conducted to determine whether the classification method described in Equations 1 and 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.

*1) **Method 1: Training with Tactic Descriptions**:* In the first experiment, we trained the classifier using the *tactic descriptions* and then tested the trained classifier against the *code snippets*. The experiment was repeated using a variety of term thresholds and classification thresholds.

*2) **Method 2: Training with Code Snippets**:* In the second experiment we trained the classifier using code snippets. Because of the time-consuming nature of finding and retrieving architectural tactics from large open-source systems, we adopted a standard 10-fold cross-validation process in which the code-snippets dataset served as both the training and testing set. In each execution, the data was partitioned by project such that in the first run nine projects, each including one related and four unrelated code-snippets, 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.

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

C. Evaluation Metrics

Results were evaluated using four standard metrics of recall, precision, f-measure, and specificity computed as follows where *code* is short-hand for *code snippets*.

Table III
A SUMMARY OF THE HIGHEST SCORING RESULTS

Tactic	Training Method	FMeasure	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

$$Recall = \frac{|RelevantCode \cap RetrievedCode|}{|RelevantCode|} \tag{3}$$

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

$$Precision = \frac{|RelevantCode \cap RetrievedCode|}{|RetrievedCode|} \tag{4}$$

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:

$$FMeasure = \frac{2 * Precision * Recall}{Precision + Recall} \tag{5}$$

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

$$Specificity = \frac{|NonRelevantCode|}{|TrueNegatives| + |FalsePositives|} \tag{6}$$

D. Results

Figure 3 reports the f-measure results for classifying classes by tactic using several combinations of threshold value. In four of the five cases, namely *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. 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.

Table III reports the optimal results for each of the tactics i.e. a result which achieved high levels of recall (0.9 or higher if feasible) while also returning as high precision as possible. The results show that in four cases the code-trained classifier recalled all of the tactic related classes, while also achieving reasonable precision. The description-trained classifier achieved recall of 1 for only two of the tactics.

Table IV
INSTANCES OF ARCHITECTURAL TACTICS IN APACHE HADOOP

Tactic	Class Count	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 server 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.

V. ROLE LEVEL LINK RECONSTRUCTION

To train a classifier to differentiate between various tactic roles, we constructed a *Role Snippets Dataset*. This dataset was a modification of the *code snippets* dataset in which each of the tactic-related code snippets was replaced by 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 code snippets to see if we could effectively retrieve classes according to their role in the project. Results from this experiment (not otherwise reported) showed that the terms used across roles in a given tactic were quite similar and so differentiation was poor. To address these problems we conducted an extensive exploratory investigation to determine how best to classify classes by roles; however we report only the final technique that was adopted. The first two steps in the process utilize the previously described classifier, while steps three to six utilize light-weight structural analysis. The heuristics of this analysis were derived through analyzing the tactic-related code found in Fault Tolerant CORBA, the Google Chromium OS and the ROSSUME robotic system. We hypothesized that utilizing class hierarchy information and class dependencies caused by method calls could improve the quality of tactic traceability.

Our approach includes includes the following steps:

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.

We explored other options for structural analysis. 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.

Unfortunately, it was not feasible to evaluate this lightweight structural approach against the previously used code snippets, as they did not carry associated structural information. We therefore conducted an initial evaluation of this approach within the richer context of the following case-study. Specific results are reported in Section VI-C.

VI. A Case Study

The goal of the case study is to reconstruct tactic-related traceability links in the Apache Hadoop software framework, a system which supports distributed processing of large datasets across thousands of computer clusters. The Hadoop library includes over 1,700 classes and provides functionality to detect and handle failures in order to deliver high availability service even in the event that underlying clusters fail.

A. Tactics in Apache Hadoop

The first step of the case study involved building an 'answer set' for evaluation purposes by manually identifying *heartbeat*, *resource pooling*, *scheduling*, *audit trail*, and *authentication* tactics in Hadoop. This 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 and eliciting feedback. As a result of the forum discussions one additional instance of the heartbeat tactic was identified. Table IV documents 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.

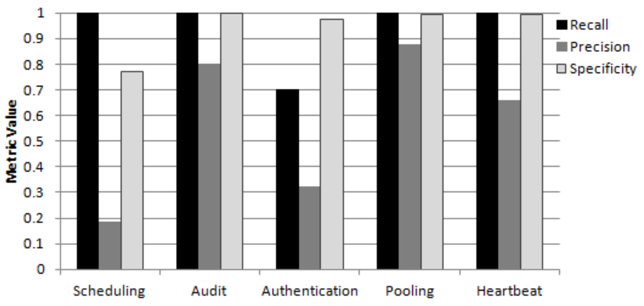


Figure 4. Results for Coarse-Grained Tactic Traceability in Hadoop

B. Reconstructing Coarse-Grained Links

The code-trained classifier developed in our previous experiments was used to classify all 1,700 classes in Hadoop according to the five targeted tactics. Based on an initial analysis of the results we established relatively low threshold levels (i.e. term threshold of 0.001 and classification threshold of 0.5) in order to achieve high recall levels.

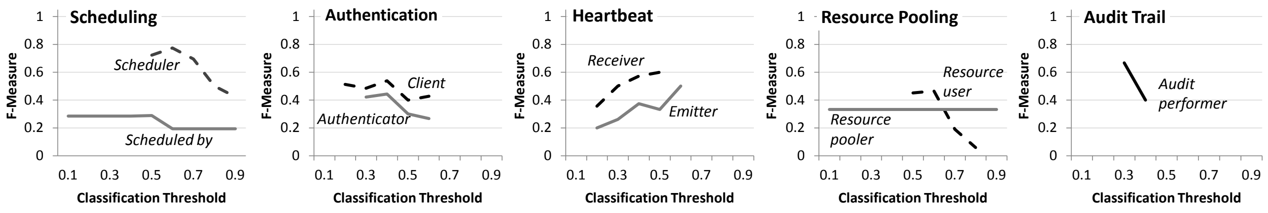


Figure 5. Results for Fine-Grained Tactic Traceability in Hadoop

Results, reported in terms of Recall, Precision, Specificity, and F-Measure are depicted in Figure 4 and show that we were able to correctly reject approximately 77-99% of the unrelated code snippets in each of the cases. In four cases, namely *scheduling*, *audit*, *resource pooling*, and *heartbeat* we were able to recall all of the related code snippets; however for *authentication* we were only able to recall 70% of the related code snippets. An analysis of the missing snippets suggested that the training set did not provide sufficient coverage of the concept, and therefore extending the scope of the training set might mitigate this problem. In all five cases precision ranged from 19% to 87%.

C. *Role-grained Trace Links*

The augmented role-grained classifier, described in Section V, was used to classify the tactic-related classes by role, with the structural analysis performed utilizing the *Understand* tool (scitools.com). Based on initial analysis of results, the classification threshold was set at 0.5. Results are reported in Figure 5. In each case, the fine-grained classifier was able to classify one dominant role better than the other one. For example 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.

To illustrate these results we present a more detailed example of one of the heartbeat instances in Hadoop. Figure 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 6, roles are ordered according to probability for each class, and if all probabilities fall below the classification threshold 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**.

D. *Trace Reconstruction*

In Figure 7 we show how the subset of role-classified classes are mapped to specific roles in a tTIM, while other unclassified classes 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."

E. *Utilizing Coarse-Grained Links*

An important, yet often unexplored research question addresses the issue of whether automatically reconstructed traceability links are good enough for use. We therefore designed an experiment to evaluate the usefulness of the generated 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 8 which utilizes traceability to provide useful architectural information.

Table V(a) reports the numbers of successfully generated notifications (true positives), unnecessary notifications (false positives), missed notifications (false negatives), and 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 (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

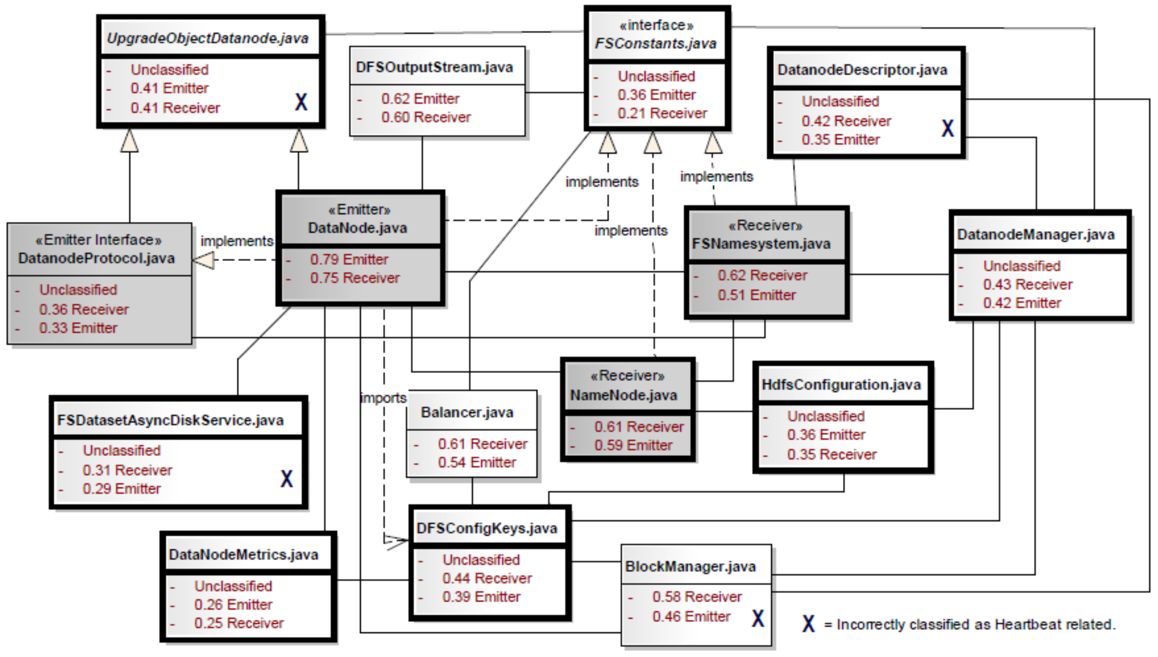
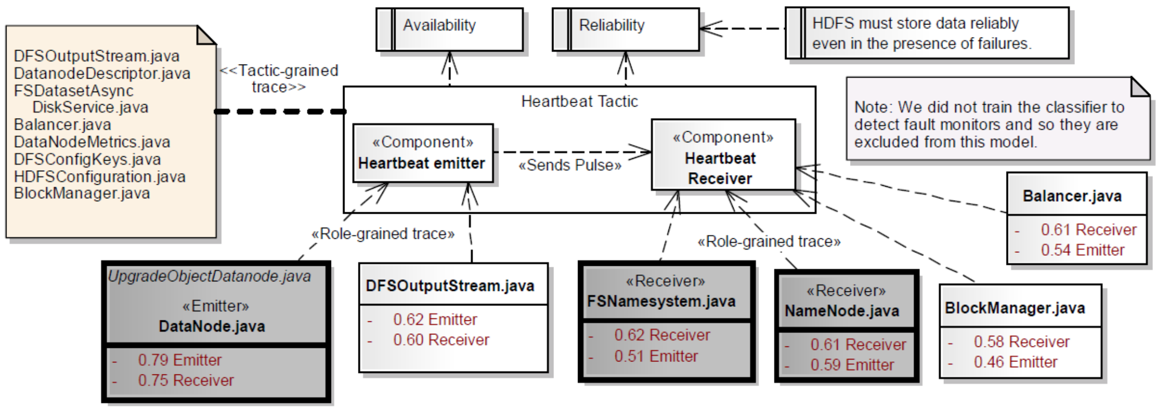


Figure 6. Reverse Engineered Role-Grained Traces for a Heartbeat Tactic in Hadoop



Availability
Reliability
HDFS must store data reliably even in the presence of failures.
Heartbeat Tactic
«Component» Heartbeat emitter
«Sends Pulse»
«Component» Heartbeat Receiver
Note: We did not train the classifier to detect fault monitors and so they are excluded from this model.
<<Tactic-grained trace>>
«Role-grained trace»
DFSOutputStream.java DatanodeDescriptor.java FSDatasetAsync DiskService.java Balancer.java DataNodeMetrics.java DFSConfigKeys.java HDFSConfiguration.java BlockManager.java

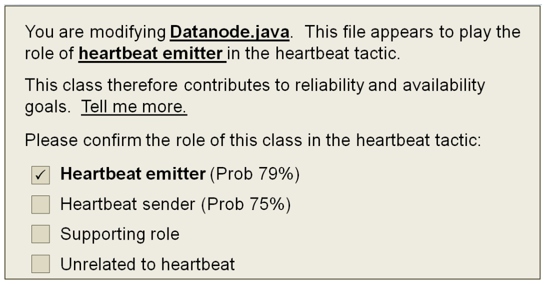
Figure 7. Trace Reconstruction through Mapping Classified Classes at both Tactic and Role Granularities to a tactic Traceability Information Model

0.93 in all cases except for the *scheduling* tactic; however precision ranged from 0.35 to 0.96. Table V(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. This relevance feedback results in improved recall, precision, and specificity for all five tactics. In fact all metrics are over 0.92 except the precision for the *scheduling* tactic which remains at 0.73. These results demonstrate the viability of our approach for supporting architectural preservation during the software maintenance effort.

VII. Threats to Validity

There are several threats to validity that may have impacted our work. One threat is related to the correctness of the training and test sets used in this study. The task of locating and retrieving over 100 samples of tactic-related code snippets, and 400 unrelated 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 search process we followed, including the careful peer-review process, gave us confidence that each of the code snippets was a correct example of its relevant tactic.

Another 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



You are modifying **Datanode.java**. This file appears to play the role of **heartbeat emitter** in the heartbeat tactic.

This class therefore contributes to reliability and availability goals. Tell me more.

Please confirm the role of this class in the heartbeat tactic:

[✓ **Heartbeat emitter** (Prob 79%)]

Heartbeat sender (Prob 75%)

Supporting role

Unrelated to heartbeat

Figure 8. A Notification Message Generated from an Automated Trace

partially mitigated this risk through locating tactics using searching, browsing, and expert opinion. 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 16 different open source systems. The leave-one-out 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 subsystems 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.

VIII. Related Work

Several researchers have developed techniques for managing design decisions [23], [27], [28], or capturing and tracing architectural knowledge [8], [21]. However, these approaches are manual in nature and fail to provide guidance on how to create and manage the potentially large number of traceability links needed to make the design knowledge available to a wide group of project stakeholders during software maintenance activities. Other researchers have developed techniques that are designed to increase program comprehension by reconstructing various architectural views [22], [31], however these techniques tend to emphasize reconstruction of the high-level structure of the system and not the underlying architectural decisions.

The significant body of prior work in the area of design pattern detection [3], [14], [29]; provided a basis for the structural analysis techniques described in our approach; however, the problem of design pattern detection is fundamentally different from the problem of tactic detection because a single tactic can often be implemented in numerous different ways. Therefore we cannot primarily rely upon structural analysis techniques.

Table V
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

(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

Finally, our work is a special case of automated trace retrieval, which can be used to generate traces between documents and source code. There are many different approaches including the vector space model, probabilistic network (PN) models, latent semantic analysis (LSA) [2], [17], and orthogonal IR based models [25]; however these techniques assume that the source of the trace, i.e. the requirement or use-case, is unique in each project, and therefore utilize basic information retrieval techniques. Finally, in our own prior work we utilized the machine learning techniques described in this paper to trace regulatory codes to requirements [9], and to differentiate between functional and non-functional requirements [10]; however neither of these techniques incorporated the richer contextual information provided by the tactic Traceability Information Models.

IX. Conclusions

This paper has presented a technique for automating the reconstruction of traceability links between classes and architectural tactics. Integrating the concept of tTIMs with existing notions of trace retrieval and classification introduces a novel approach to tracing architectural concerns. It minimizes the human effort required to establish traceability and produces traces which can be used to support critical software engineering tasks such as software maintenance, in order to help mitigate the pervasive problem of architectural erosion.

Future work will investigate additional tactics, refine the trace reconstruction algorithms, and evaluate our approach against a broader set of software systems. In addition, we are developing an Eclipse plug-in which will allow us to study this, and related techniques, within the context of an ongoing project.

Acknowledgments

The work in this paper was partially funded by the US National Science Foundation grant # CCF-0810924.

REFERENCES

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

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

[3] G. Antoniol, G. Casazza, M. D. Penta, and R. Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59(2):181–196, 2001.

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

[5] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Adison Wesley, 2003.

[6] B. Berenbach, D. Gruseman, and J. Cleland-Huang. Application of just in time tracing to regulatory codes. In *Proceedings of the Conference on Systems Engineering Research*, 2010.

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

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

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

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

[11] J. Cleland-Huang, G. Zemont, and W. Lukasik. A heterogeneous solution for improving the return on investment of requirements traceability. In *RE*, pages 230–239, 2004.

[12] D.E.Perry and A.L.Wolf. Foundations for the study of software architecture. *SIGSOFT Software Eng. Notes*, 17(4):40–52, 1992.

[13] A. Egyed, S. Biffl, M. Heindl, and P. Grünbacher. Determining the cost-quality trade-off for automated software traceability. In *ASE*, pages 360–363, 2005.

[14] F. A. Fontana, M. Zanoni, and S. Maggioni. Using design pattern clues to improve the precision of design pattern detection tools. *Journal of Object Technology*, 10:4: 1–31, 2011.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns, software engineering, object-oriented programming*.

[16] O. Gotel and A. Finkelstein. Extended requirements traceability: Results of an industrial case study. In *RE '97: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, page 169, Washington, DC, USA, 1997. IEEE Computer Society.

[17] M. Grechanik, K. S. McKinley, and D. E. Perry. Recovering and using use-case-diagram-to-source-code traceability links. In *ESEC/SIGSOFT FSE*, pages 95–104, 2007.

[18] R. Hanmer. *Patterns for Fault Tolerant Software*. Wiley Series in Software Design Patterns, 2007.

[19] A. Jalaji, R. Goff, N. Jones, and T. Menzies. Making sense of text: Identifying nonfunctional requirements early. *Technical Report, West Virginia University*.

[20] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *WICSA*, pages 109–120, 2005.

[21] A. Jansen, J. S. van der Ven, P. Avgeriou, and D. K. Hammer. Tool support for architectural decisions. In *WICSA*, page 4, 2007.

[22] R. Koschke. Architecture reconstruction. In *ISSSE*, pages 140–173, 2008.

[23] P. Kruchten, R. Capilla, and J. C. Dueas. The decision view's role in software architecture practice. *IEEE Software*, 26(2):36–42, 2009.

[24] M. Mirakhorli and J. Cleland-Huang. 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.

[25] Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk and Andrea De Lucia. On integrating orthogonal information retrieval methods to improve traceability link recovery. In *International Conferences on Software maintenance*, 2011.

[26] M. Mirakhorli and J. Cleland-Huang. A pattern system for tracing architectural concerns. In *Proc. of the Pattern Languages of Programming*, 2011.

[27] M. Mirakhorli and J. Cleland-Huang. Tracing architectural concerns in high assurance systems: (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 908–911, 2011.

[28] M. Mirakhorli and J. Cleland-Huang. Using tactic traceability information models to reduce the risk of architectural degradation during system maintenance. In *Proc. of the International Conf. on Software Maintenance, ICSM*, pages 123–132, 2011.

[29] N. Pettersson, W. Löwe, and J. Nivre. Evaluation of accuracy in design pattern occurrence detection. *IEEE Trans. Software Eng.*, 36(4):575–590, 2010.

[30] J. van Gurp, S. Brinkkemper, and J. Bosch. Design preservation over subsequent releases of a software product: a case study of baan erp: Practice articles. *J. Softw. Maint. Evol.*, 17:277–306, July 2005.

[31] H. Verjus, S. Cîmpan, A. Razavizadeh, and S. Ducasse. Beeeye: A framework for constructing architectural views. In *ECSA*, pages 376–383, 2010.