Detecting, Tracing, and Monitoring Architectural Tactics in Code

Mehdi Mirakhorli* *Member, IEEE* Jane Cleland-Huang† *Member, IEEE*
*Department of Software Engineering
Rochester Institute of Technology, Rochester, NY
mehdi@se.rit.edu
†School of Computing
DePaul University, Chicago, IL 60604
jhuang@cs.depaul.edu

Abstract—Software architectures are often constructed through a series of design decisions. In particular, architectural tactics are selected to satisfy specific quality concerns such as reliability, performance, and security. However, the knowledge of these tactical decisions is often lost, resulting in a gradual degradation of architectural quality as developers modify the code without fully understanding the underlying architectural decisions. In this paper we present a machine learning approach for discovering and visualizing architectural tactics in code, mapping these code segments to Tactic Traceability Patterns, and monitoring sensitive areas of the code for modification events in order to provide users with up-to-date information about underlying architectural concerns. Our approach utilizes a customized classifier which is trained using code extracted from fifty performance-centric and safety-critical open source software systems. Its performance is compared against seven off-the-shelf classifiers. In a controlled experiment all classifiers performed well; however our tactic detector outperformed the other classifiers when used within the larger context of the Hadoop Distributed File System. We further demonstrate the viability of our approach for using the automatically detected tactics to generate viable and informative messages in a simulation of maintenance events mined from Hadoop's change management system.

Index Terms—Architecture, traceability, tactics, traceability information models

1 Introduction

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, availability, and performance. Architects often integrate known architectural frameworks, styles, and tactics into their designs in order to address specific quality concerns. For example, performance issues in a data-intensive application might be addressed using *lazy-load* [1], while availability of a critical component could be monitored by use of *ping-echo* or the *heart-beat* tactic [2].

Unfortunately, the problem of architectural degradation is common place [1], [3], [4]. Architectures which are carefully designed during initial phases of the project, gradually degrade over time as developers perform maintenance activities and introduce new features to the system [5], [6]. This problem is at least partially caused by developers' lack of underlying architectural knowledge – if they are not informed of the design decisions that influenced or drove the initial implementation, then they are unlikely to preserve those decisions during the modification process [7].

In practice, many software projects fail to fully document architectural decisions, providing only high-level lists of decisions. However, several solutions have been proposed. Burge et al. [8] and Kruchten [9] used design rationales to document architectural design decisions, their justifications, alternatives, and tradeoffs. Other similar approaches include the Architecture Design Decision Support System (ADDSS) [10], Process based Architecture Knowledge Management Environment (PAKME) [11], and Architecture Rationale and Element Linkage (AREL) [12]. However, such approaches generally document rationales at the design level, and fail to provide explicit traceability down to the code level in which the decisions have been implemented [13]. Existing design rationale systems therefore provide only limited support for keeping developers informed of underlying design decisions during the code maintenance process.

To address this problem we present a solution for detecting architectural tactics in code, tracing them to requirements, and visualizing them in a way that helps a developer to understand underlying design decisions. We use machine learning to train a classifier to detect the presence of architectural tactics, and then to map relevant parts of the code to *Tactic Traceability Patterns* (tTPs) so that architecturally significant code can be monitored and code modifications can trigger informational notifications to developers. As such, our approach is designed to address the problem of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2015.2479217, IEEE Transactions on Software Engineering

TABLE 1
Architectural Tactics Discovered through Inspecting Documentation and Code for 20 Open Source Performance-centric and/or Safety-critical Systems

	Fault tolerant, performance-centric software systems	Active Repl.	Audit Trail	Authentication	Authorization	CRC	Encryption	Fault Detection	Heartbeat	Passive Repl.	Permiss. Check	Process Monitor	Recovery	Rem. Service	Resource Pool.	Scheduling	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.

architectural degradation through keeping developers informed of underlying architectural decisions, even when those decisions are not formally documented.

We focus our efforts on detecting the class of architectural decisions referred to as architectural tactics [2], [14], [15]. Tactics represent re-usable solutions for satisfying specific quality concerns, and are defined by Bachman et al. as "a means of satisfying a quality-attribute-response measure by manipulating some aspects of a quality attribute model through architectural design decisions" [16]. Architectural tactics come in many different shapes and sizes providing solutions for a wide range of quality concerns. For example, a system with high reliability requirements might implement the *heartbeat* tactic [2] to monitor availability of a critical component, or the *voting* tactic [2] to increase fault tolerance through integrating and processing information from a set of redundant components.

The work described in this paper builds upon our prior work [13], [17] which described our approach for detecting architectural tactics and for monitoring them in the code. However, we make several additional non-trivial contributions in this paper. First and foremost, the earlier work evaluated our approach against five architectural tactics across code extracted from ten projects. In this paper we conduct a far more extensive evaluation of ten different tactics against code taken from 50 projects – thereby demonstrating far stronger generalizability. Secondly, we perform a comparative analysis of the 'home-grown' tactic-detector used in our prior work against six well-known classifiers as well as a 'super-classifier' which utilizes a voting scheme to integrate individual results. We perform this analysis as a controlled experiment and also within the context of a large-sized open source project. Third, we describe our Archie tool [18], which demonstrates the application of our approach within the context of an integrated development environment (IDE). Finally, for reproducibility purposes, we release the datasets from our controlled study via the CoEST.org dataset directory [1].

The remainder of this paper is laid out as follows. Section 2 provides a more detailed explanation of architectural tactics, and introduces the concept of tTPs. Section 3 describes our tactic detector and the studies we conducted to comparatively evaluate it against off-the-shelf classifiers. Section 4 describes a case study in which the tactic detector and other classifiers are utilized and evaluated within the context of the Apache Hadoop framework. Section 5 presents our Archie tool and describes usage scenarios that could lead to architectural preservation within the context of a series of maintenance activities. The paper concludes with a discussion of threats to validity in Section 6, related work in Section 7, and finally conclusions in Section 8.

2 Architectural Tactics

Architectural tactics come in many different shapes and sizes and describe solutions for a wide range

1. http://coest.org/mt/27/150

of quality concerns [2]. 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 [19]. In early phases of our project we manually inspected the code and supporting documentation of twenty performance-centric, fault-tolerant, open-source systems, searching for evidence of 16 well-documented architectural tactics. The occurrence of these tactics across the studied systems is reported in Table 1 and clearly highlights the pervasive nature of architectural tactics in the examined systems.

2.1 Tactics Selected for this Study

For purposes of our ongoing study we selected ten commonly occurring architectural tactics representing three different quality concerns. These included five security tactics, namely *audit*, *authenticate*, *HMAC*, *Secure Session Management* and *RBAC*; two reliability tactics, namely *heartbeat* and *CheckPoint* and three performance tactics *Resource Pooling*, *Resource Scheduling* and *Asynchronous Invocation*. The ten selected tactics are defined as follows [2]:

1-Asynchronous Communication: In synchronous communication a method is invoked and all other operations are blocked until the call is completed. To decrease response time and/or increase throughput, asynchronous communication and method invocation is used to return control to the calling application before obtaining a response.

2-Audit trail: A copy of each transaction and associated identifying information is maintained. This audit information can be used to recreate the actions of a malicious user, and also to support functions such as system recovery and non-repudiation.

3-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.

4-Checkpoint/rollback Uses a checkpoint to record system state during normal execution and in case of failure uses this information to recover the system to a previously safe state. The logging is done either periodically or in response to specific events.

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.

6-HMAC: Provides 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 enables integrity assurances to detect accidental and intentional message changes, while at the same time authenticity assurances affirm the message's origin.

7-RBAC: User/Process Authorization is used to ensure 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, for example based on user roles/classes or through specific policies. Two major types of authorization therefore include Role Based Access Control (RBAC) and Policy Based Access Control (PBAC).

8-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.

9-Scheduling: Resource contentions are managed through scheduling policies such as FIFO (First in First out), fixed-priority, and dynamic priority scheduling.

10-Secure Session: Allows an application to only require the users to authenticate once to 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.

2.2 Tactic Traceability Patterns

Our approach to tracing architectural tactics builds upon the fundamental concept of the tTP described in our prior work [17], [20]. The tTP 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 [21], [22], NASA robots [23]–[26], and also performance centric systems such as Google Chromium OS [17], [27]. Each tTP 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 tTP 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 1(e), includes the primary roles of *receiver*, *emitter*, and *fault monitor*. Additional roles, not shown in this figure but described in our earlier work [17], include parameters such as the *heart beat rate* and the *heartbeat checking interval*. In addition to roles, the tTP also includes a set of reusable, semantically-typed traceability links. These include internal links such as

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2015.2479217, IEEE Transactions on Software Engineering

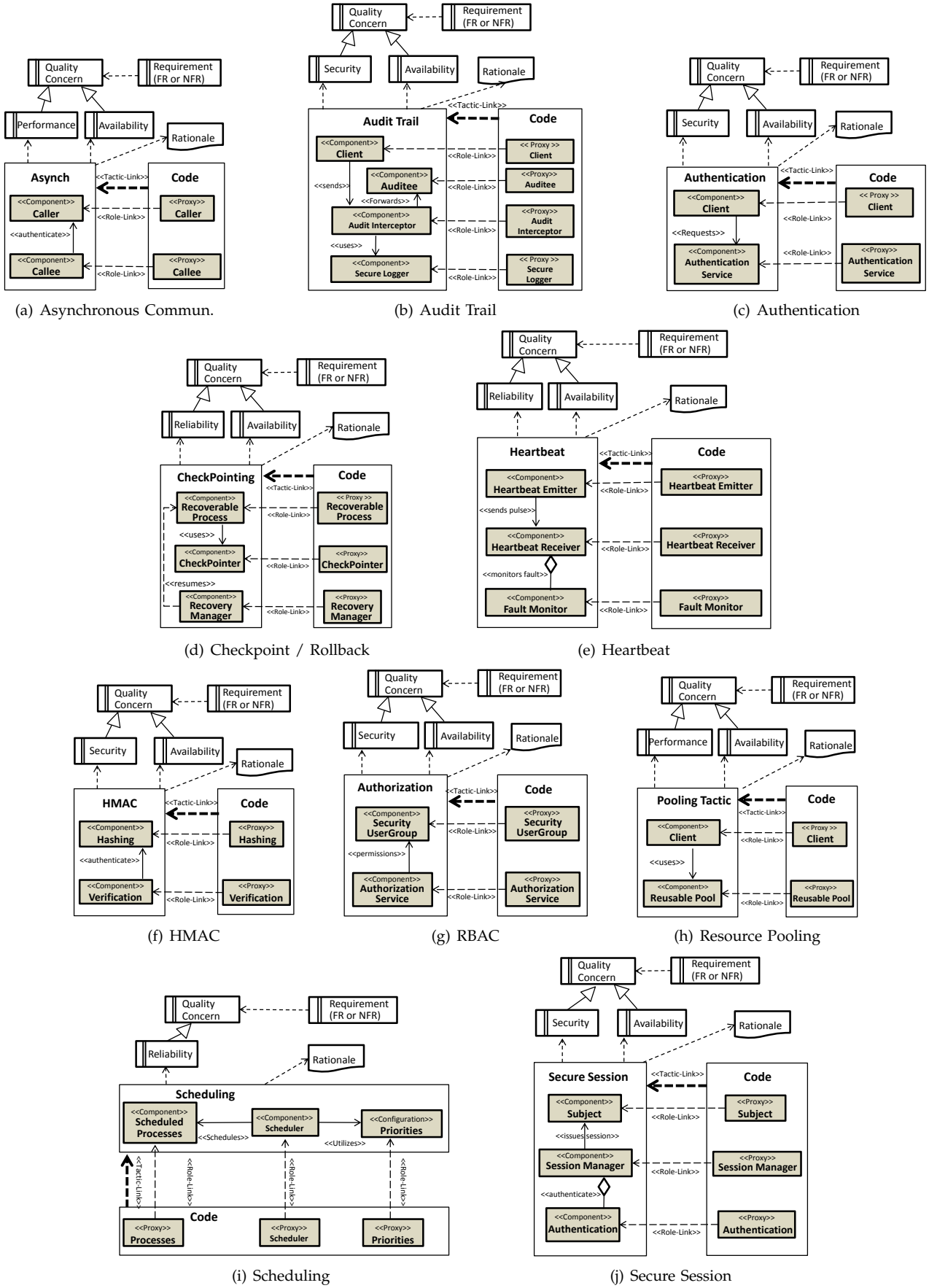

Fig. 1. Tactic Traceability Patterns (tTPs) for Ten Architectural Tactics

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2015.2479217, IEEE Transactions on Software Engineering

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 tTP 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 one or more elements in the architecture and/or code onto the proxy in order to establish traceability. Once a tTP has been instantiated and these mappings established, all of the traceability information embedded in the tTP is automatically inherited by the project. As a result, traceability is established from code all the way back to quality-related concerns and requirements. Because of the inherent reuse of trace links, tTPs have been shown to reduce the cost and effort of traceability [17].

3 Detecting Tactics in Source Code

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 [28], tactics are described in terms of roles and interactions [2]. 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 [28] in which the receiver registered as a listener to the emitter *found in the Amalgam system*, (iii) the decorator pattern [28] 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. Numerous studies have shown that programmers tend to use meaningful terms to name variables, methods, and classes, and also often provide meaningful comments which offer insights into the purpose of the code [29], [30].

3.1 Classifiers

We comparatively evaluated the efficacy of six different classifiers for the task of detecting architectural tactics in code. These included the tactic detector (TD), support vector machine (SVM), C.45 decision tree (implemented as J48 in Weka), bayesian logistic regression (BLR), AdaBoost, rule learning with SLIPPER, and bagging. These classifiers were selected as they have all been used to solve similar problems in the past. As the tactic detector is our own homegrown technique we describe it in greater detail than the other standard solutions.

3.1.1 Tactic Detector

The tactic detector [31] is a customized version of our previously developed NFR-Classifier [32]. It involves three phases of preparation, training, and detection which are defined as follows:

Preparation All data are preprocessed using standard information retrieval techniques. These include removal of non-alpha-numeric characters, stemming words to their morphological roots, and removal of 'stop' words, i.e. commonly occurring words such as 'this' and 'shall' which are not helpful for classification purposes. The remaining terms are then transformed into a vector of terms.

Training The training phase 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 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.

The following lines of code are extracted from a code snippet used to train the 'audit trail' classifier. We can observe that the programmer utilized obviously meaningful terms, such as *Audit*, in the code.

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

The tactic detector is defined more formally as follows. 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:

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

For illustrative purposes, Table 2 depicts the top ten indicator terms learned for each of our ten tactics when running experiments described in Section 3.2.2 of this paper. The terms are depicted in their stemmed form.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2015.2479217, IEEE Transactions on Software Engineering

TABLE 2
Indicator terms for each of the ten architectural tactics learned during training

Tactic Name	Code trained indicator terms
Asynch	event, asynchron, async, method, invoc, param, object, valu, except, invok
Audit Trail	audit, trail, wizard, pwriter, lthread, log, string, categori, pstmt, pmr
Authentication	authent, credenti, challeng, kerbero, auth, login, otp, cred, share, sasl
CheckPoint	checkpoint, file, transact, commit, set, log, index, info, directori, rollback
Heartbeat	heartbeat, ping, beat, heart, hb, outbound, puls, hsr, period, isonlin
HMAC	kei, hmac, sha, algorithm, hash, data, param, messag, mac, authent
RBAC	role, permiss, user, access, resourc, code, set, secur, param, author
Resource Pooling	pool, thread, connect, sparrow, nbp, processor, worker, timewait, jdbc, ti
Scheduling	schedul, task, prioriti, prcb, sched, thread, rtp, weight, tsi
Session	session, user, set, http, request, secur, id, valu, url, util

Detection During this 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 a 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.

3.1.2 Support Vector Machine

A Support Vector Machine (SVM) is a powerful classifier which has been used for classification purposes across a wide variety of problems including bug prediction, text classification and feature selection [33]. 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 accomplished through a technique known as the "kernel method" [34]. SVM performs well on data with a high dimensional input space. It can also handle sparse vectors. Both of these characteristics are found in text classification problems [33] [35].

3.1.3 Classification by Decision Tree (C.45)

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. In a decision tree, the internal nodes represent test conditions while leaf nodes represent categories. In our case there are two categories of tactical and non-tactical. The attributes chosen to build the tree are based on information gain theory, meaning that the internal nodes of the decision tree are constructed from the attributes that provide maximum information while the leaf node predicts a category or class [36].

3.1.4 Bayesian Logistic Regressions (BLR)

Bayesian Logistic Regression models are also often used for text classification purposes [37]. In general, results from BLR have been shown to be as effective as SVM classifiers. BLR is a linear model in which predictions are transformations of a weighted sum of the features modeling the probability of categorical outcomes via a logistic link function. Model fitting involves inferring the best values from training data while accounting for correlations amongst features. For our purposes we estimate the probability that a particular source file represents a specific architectural tactic. A threshold value is established for making this determination.

3.1.5 AdaBoost

Boosting uses a voting mechanism to combine results from many relatively weak and inaccurate classifiers with the goal of delivering a high-performing one. We used the AdaBoost algorithm proposed by Freund and Schapire [38], as it has been widely used to solve several similar software engineering problems [31]. The weight assigned to each classifier (in Weka) is equal to *log(1/Beta)*$where$*beta = error/(1-error)*. Adaboost increases the weight of cases which are hard to classify at each iteration. Boosting has been shown to be effective for predicting hard to classify cases.

3.1.6 Ensembled Rule Learning: SLIPPER

The Simple Learner with Iterative Pruning to Produce Error Reduction (SLIPPER) approach [39] is a standard rule-learning algorithm based on confidence-rate boosting. It is frequently used for text classification problems. 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 between this approach and 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 from which a binary classifier is ultimately constructed.

3.1.7 Bagging

Bagging [40] works in the same way as boosting but utilizes a simpler way for generating the training set. It trains individual classifiers on a random sampling of the original training set. A majority vote on the classification results define the final results. Bagging has been shown to be effective for "unstable" learning algorithms such as decision trees where small changes in the training set result in large changes in predictions [41].

3.2 Evaluating the Classifiers

To perform an initial evaluation of the classifiers we designed an experiment in which each classifier was trained and tested against a dataset of architectural code snippets. We describe the construction of the dataset and the experimental evaluation below.

3.2.1 Training Set

For each of the ten tactics, we identified 50 open-source projects in which the tactic was implemented. For each project we performed an *architectural biopsy* to retrieve a source file in which the targeted tactic was implemented and also retrieved one randomly selected non-tactical file. Using this data we built a balanced training set for each tactic which included 50 tactic-related snippets and 50 non-tactical ones.

Our training set was limited to java code. Architectural tactics were discovered as follows:

- **Direct Code Search:** The source code search engine Koders was used to search for the tactic. The search query for each tactic was composed from keywords used in descriptions of the tactic found in textbooks, articles, and white papers. All returned code was reviewed by two team members to determine whether it was relevant (i.e. related to the current architectural tactic) or not.
- **Indirect Code Search:** Project-related documents, such as design documents, online forums, etc. were searched for references and pointers to architectural tactics. This information was then used to identify and retrieve relevant code. Once again, all retrieved code was carefully reviewed to ensure that it was relevant.
- **"How to" examples:** Online materials and libraries (e.g. MSDN, stack overflow) were searched for concrete examples of implemented architectural tactics.

We discuss threats to validity concerning the selection and the tactic-representativeness of the identified code-snippets in Section 6.

3.2.2 N-Fold Cross-Validation Analysis

A standard n-fold cross-validation experimental design was adopted in which each dataset of tactics was divided into five equal buckets - each bucket containing tactical and non-tactical code samples from 20% of the projects.

The six classifiers were then individually trained using four of the buckets and then tested against the remaining bucket. This process was repeated five times until each bucket had been classified by each classifier. The process was repeated for each of the ten tactics.

3.3 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*.

$$Recall = \frac{|RelevantCode \cap RetrievedCode|}{|RelevantCode|} \quad (3)$$

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

$$Precision = \frac{|RelevantCode \cap RetrievedCode|}{|RetrievedCode|} \quad (4)$$

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:

$$F - measure = \frac{2 * Precision * Recall}{Precision + Recall} \quad (5)$$

Finally, specificity measures the fraction of unrelated and unclassified code snippets that are correctly rejected by our classifier. It is computed as:

$$Specificity = \frac{|NonRelevantCode|}{|TrueNegatives| + |FalsePositives|} \quad (6)$$

3.4 Parameter Optimization

For each classifier we conducted a series of systematic n-fold experiments to identify the best configuration parameters. For example, Figure 2 reports the F-measure results for classifying classes by Tactic Detector using several combinations of threshold value. Similar experiments were conducted for each classifier resulting in the configuration values shown in Appendix A. We report these values for reproducibility purposes.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2015.2479217, IEEE Transactions on Software Engineering

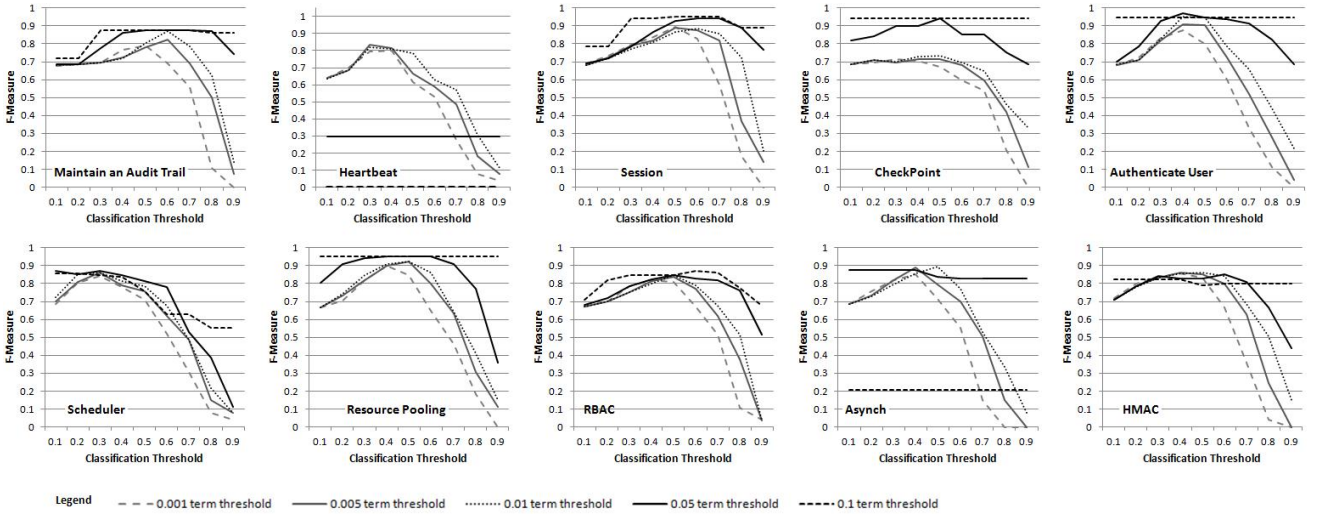

Fig. 2. Results for Detection of Tactic-related Classes at various Classification and Term Thresholds for Ten Different Tactics

TABLE 3
Results from N-Fold Cross Validation Detection of Architectural Tactics

	SVM			Slipper			J48			Bagging			AdaBoost			Bayesian			Tactic Det.		
	Pr	Rec	FM	Pr	Rec	FM	Pr	Rec	FM	Pr	Rec	FM	Pr	Rec	FM	Pr	Rec	FM	Pr	Rec	FM
Audit	0.96	0.46	0.62	0.85	0.78	0.81	0.85	0.85	0.85	0.88	0.88	0.88	0.85	0.85	0.85	0.94	0.91	**0.92**	0.84	0.92	0.88
Authenticate	0.91	0.58	0.71	0.96	0.94	0.95	0.98	0.98	0.92	1.00	0.92	0.96	0.98	0.98	0.94	1.00	0.80	0.89	0.96	0.98	**0.97**
Heartbeat	0.91	0.62	0.74	0.84	0.84	0.84	0.77	0.88	0.82	0.89	0.84	0.87	0.91	0.86	**0.89**	0.92	0.70	0.8	0.77	0.92	0.84
Pooling	0.97	0.66	0.79	0.94	0.96	0.95	0.94	0.96	0.95	0.94	0.94	0.94	0.98	0.96	**0.97**	0.94	0.96	0.95	0.92	0.98	0.95
Scheduler	0.98	0.88	0.93	0.88	0.92	0.9	1.00	0.98	**0.99**	1.00	0.98	**0.99**	1.00	0.98	**0.99**	0.96	0.98	0.97	0.86	0.88	0.87
Asynch	0.58	0.92	0.71	0.96	0.96	0.96	0.96	0.98	**0.97**	0.96	0.98	**0.97**	0.96	0.98	**0.97**	0.8	0.78	0.79	0.95	0.84	0.89
HMAC	0.78	0.76	0.77	0.96	0.96	0.96	0.96	1.00	**0.98**	0.94	0.98	0.96	0.96	1.00	**0.98**	0.95	0.84	0.89	0.91	0.82	0.86
RBAC	0.83	0.60	0.70	0.91	0.86	0.89	0.92	0.88	**0.90**	0.92	0.88	**0.90**	0.92	0.88	**0.90**	0.92	0.88	**0.90**	0.86	0.88	0.87
Session	0.63	0.8	0.71	0.91	0.98	0.94	0.91	0.98	0.94	0.91	0.96	0.93	0.91	0.98	0.94	0.76	0.87	0.81	0.91	1.00	**0.95**
Checkpoint	0.62	0.42	0.5	0.9	0.92	0.91	0.94	0.94	0.94	0.94	0.94	0.94	1	0.94	0.97	1.00	0.97	**0.99**	0.94	0.94	0.94
Averages	0.82	0.67	0.72	0.91	0.91	0.91	0.92	0.94	0.93	0.94	0.93	0.93	0.95	0.94	0.94	0.92	0.87	0.89	0.89	0.92	0.92
	0 win			0 wins			4 wins			3 wins			6 wins			3 win			2 wins		

3.5 Results

Results from the experiment are reported in Table 3. A comparison of F-measure scores show that no clear winner emerged. AdaBoost returned the best results, either outrightly winning or tying for first place for five of the tactics; J48 returned best results in four cases; Bagging and Bayesian Regression Modeling won three contests each; while the Tactic Detector won two. In contrast neither SVM nor SLIPPER came first in any of the tactics, and SVM clearly underperformed the other classifiers. The Friedman non-parametric Statistical test was applied on the results of all the methods excluding SVM which was clearly non-competitive. The test results indicated that the difference between the classifiers is not statistically significant. (p-value of 0.05). Based on these results we conclude that six of the classifiers perform equivalently for the task of tactic detection.

4 Apache Hadoop Case Study

The purpose of our work is to build a classifier which can detect architectural tactics in large-scale projects. In this section we therefore applied the classifiers trained through the controlled experiment, to the task of detecting architectural tactics in the Hadoop Distributed File System (HDFS) [42]. Apache Hadoop is a framework 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.

4.1 Tactics in Apache Hadoop

The first step of the case study involved determining which of the tactics under evaluation occurred in the Hadoop code, and then identifying specific classes in which they were implemented. This task was accomplished by (i) reviewing the available Hadoop literature [42] to look for 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 the Koders search engine to search for relevant code using key terms. In order to reduce bias, this task

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2015.2479217, IEEE Transactions on Software Engineering

TABLE 4
Instances of Architectural Tactics in Apache Hadoop

Tactic	Classes	Explanation	Package Name or Subsystem
Asynchronous Communication	13	Handles communication with all the NodeManagers and provides asynchronous updates on getting responses from them	mapreduce & datanode
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.
CheckPoint	32	Periodic checkpoints of the namespace to keep NameNode and Backup NameNode in synch and help Namenode restart.	HDFS
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
Hash Based Method Authentication (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.
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
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.

TABLE 5
Comparative Evaluation ofVarious Classifiers for Detecting Architectural Tactics in Hadoop

	SVM			Slipper			J48			Bagging			AdaBoost			Bayesian			Tactic Det.			Voting		
	Pr	Rec	FM	Pr	Rec	FM	Pr	Rec	FM	Pr	Rec	FM	Pr	Rec	FM	Pr	Rec	FM	Pr	Rec	FM	Pr	Rec	FM
Audit	0.08	0.29	0.13	0.02	0.29	0.04	0.03	0.29	0.06	1.00	0.29	0.44	0.03	0.29	0.06	0.04	0.5	0.07	1.00	0.71	**0.83**	0.67	0.5	0.57
Authenticate	0.14	0.52	0.22	0.16	0.61	0.26	0.57	0.59	0.58	0.58	0.56	0.57	0.17	1.00	0.30	0.15	0.37	0.21	0.61	0.70	**0.66**	0.47	0.66	0.55
Heartbeat	0.07	0.11	0.09	0.31	0.59	0.41	0.22	1.00	0.36	0.50	1.00	0.67	0.35	0.96	0.51	0.07	0.04	0.05	0.66	1.00	**0.79**	0.57	0.96	0.71
Pooling	0.71	0.11	0.19	0.13	0.44	0.20	0.89	0.97	**0.93**	0.88	1.00	**0.93**	0.87	0.87	0.87	0.16	0.33	0.22	0.88	1.00	**0.93**	0.89	0.96	0.92
Scheduler	0.36	0.63	0.46	0.65	0.2	0.30	0.64	0.87	0.74	0.65	0.89	0.75	0.66	0.77	0.71	0.32	0.78	0.46	0.65	0.94	**0.77**	0.68	0.89	**0.77**
Asynch	1.00	0.72	**0.84**	0.19	0.44	0.26	1.00	0.72	**0.84**	1.00	0.72	**0.84**	0.82	0.50	0.62	0.00	0.00	0.00	1.00	0.72	**0.84**	1.00	0.72	**0.84**
HMAC	0.09	0.63	0.15	0.12	0.5	0.19	0.12	0.57	0.2	0.12	0.57	0.2	0.12	0.57	0.2	0.13	0.71	**0.22**	0.06	0.86	0.11	0.12	0.57	0.20
RBAC	0.12	0.13	0.12	0.19	0.49	0.27	0.2	0.21	0.2	0.42	0.21	0.28	0.35	0.28	0.31	0.03	0.13	0.05	0.31	0.97	**0.48**	0.69	0.23	0.35
Session	0.07	0.11	0.09	1.00	1.00	**1.00**	0.84	0.84	0.84	0.84	1.00	0.91	0.84	0.84	0.84	0.09	0.31	0.14	0.84	0.84	0.84	0.84	1.00	0.91
Checkpoint	0.29	0.35	0.32	1.00	1.00	**1.00**	1.00	0.94	0.97	1.00	0.94	0.97	1.00	0.97	0.99	0.12	0.68	0.21	1.00	1.00	**1.00**	1.00	0.94	0.97
	1 win			2 wins			2 wins			2 wins			0 wins			1 win			8 wins			2 wins		

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 from developers. The Hadoop developers did not refute any of the tactics we had identified, but did point out one additional instance of a tactic which we then included in our reference set. As a result of these activities Table 4 documents the occurrences of the ten architectural tactics identified in Hadoop. The identified code served as the 'reference set' for the remainder of the case study. Our analysis suggested that 1,557 classes were not tactic related, 145 classes implemented one tactic only, 14 classes implemented two tactics, two classes implemented three tactics, and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2015.2479217, IEEE Transactions on Software Engineering

TABLE 6
F-measure Reported for Different Classifiers in Hadoop Case Study

	Audit	Authentication	Heartbeat	Pooling	Scheduler	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

one class implemented four tactics.

4.2 Classifying Hadoop Tactics

Given the fact that there was no significant difference between the results produced by six of the classifiers in the controlled n-fold experiment, we decided to re-evaluate all of the classifiers in the Hadoop task. The previously trained classifiers were configured as reported in Appendix A.

In addition, we introduced an additional *voting* technique, which integrated results from all seven individual classifiers. We informally evaluated several different voting schemes including majority rules, weighted voting and "minimum of three" ; however we report only the best performing voting scheme which is majority voting.

Results are reported in Table 5 and show that the classifiers behaved somewhat differently than they did for the n-fold experiment. We first discuss the results and then suggest reasons for the observed differences in performance.

In general, this classification task is more difficult than the previous one reported in Section 3.2.2, because the size of the dataset is much larger. Instead of classifying a set of 100 code snippets evenly balanced between tactical and non-tactical code samples, we need to find anywhere from 8 classes (in the case of HMAC) to 95 classes (in the case of Resource Pooling) from a potential pool of 1,700 classes. Identifying tactical code in Hadoop is therefore more akin to the proverbial needle in a haystack problem. Not surprisingly, the average F-measure score dropped from 0.9 for the n-fold experiment to 0.53 for Hadoop.

Our own tactic detector clearly outperformed the other classifiers, producing the highest F-measure results in 8 of the 10 cases. Slipper, J48, and Bagging won in 2 cases each, SVM and BRM in one, and surprisingly AdaBoost in none. The voter only won in 3 cases, presumably because the Tactic Detector performed so well as an individual classifier.

4.3 Analysis of Results

The ultimate purpose of this case study was to determine which classifier should be used for practical purposes to classify tactics in large projects such as Hadoop. Our goal was to identify a *reliable*, *easy to configure*, and *stable* solution.

A rank comparison of the classifiers, reported in Table 6, shows that the Tactic Detector performed best for 8 of the 10 architectural tactics, while in 2 cases it was outperformed by both the Bayesian and Slipper approach. Table 7 reports both the mean and median of the F-measure for each classifier. As the data are not normally distributed, in order to evaluate whether differences were statistically significant we performed a Friedman ANOVA test which is a non-parametric test for comparing the medians of paired samples.

TABLE 7
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

The Friedman test (Table 8) indicates the there is a statistically significant difference between the performance of the classifiers; however the test does not determine which one is better than the others. As the Tactic Detector exhibited the highest mean and median (Table 7) across 10 different architectural tactics in Hadoop, we performed a pairwise comparison between it and all other classification techniques.

TABLE 8
Testing Statistically Significance in Medians of Classifiers Performance

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

Table 9 reports the results of this experiment using both non-parametric and parametric tests. Uniformly

both tests indicated that the Tactic Detector is more accurate than SVM, Slipper, AdaBoost and Bayesian classification technique at statistical significance with a p-value of 0.05. Furthermore the Tactic Detector also performed better than Bagging and J48 classification methods but this conclusion with a p-value of 0.05 and confidence level of 0.95 is not statistically significant. Similarly the difference between the Tactic Detector and the voting mechanism was also not statistically significant, most likely because the voting mechanism utilizes the Tactic Detector as a voter.

It is also interesting to observe that the performance of the Tactic Detector did not change drastically between the controlled experiment and the Hadoop case study. This contrasted with the other individual classifiers, which all exhibited sharp increases in the number of false positives when used in Hadoop. This suggests that the Tactic Detector is more stable than the other tested approaches, although further studies will be needed to substantiate this claim.

Finally, techniques such as bagging, boosting, and Slipper, are non-trivial to use, and require the creation of different rule learners. In contrast the Tactic Detector is a simple, linear classifier which scales up for large-sized datasets. We were able to analyze the HDFS code and to classify all eight tactics for Hadoop in approximately 20 seconds on a laptop.

5 Architectural Preservation

In this section we discuss and evaluate the usefulness of the generated trace links. Although the tactic detector can be used as an isolated utility, we have designed it to be used primarily within an instrumented software engineering environment.

Figure 3 depicts our Eclipse plugin tool named *Archie* [18]. To support our goal of architectural preservation Archie delivers various capabilities including (1) a detection engine capable of identifying code implementing a predefined set of architectural tactics and an interactive viewer which allows a user to browse through code snippets returned by the detection engine, (2) an annotated code viewer, which highlights architecturally significant parts of the code, (3) visualization features for generating views of specific architectural tactics, their relationships to design rationales and requirements, and global views of architectural decisions, (4) features to allow a user to bypass the automated detection process and to manually mark-up sections of code as being architecturally significant, and finally (5) an event engine which constantly monitors changes to the code in the background, notifies the user when he/she starts to modify sensitive areas of the code, and displays information about the underlying architectural decisions. A more complete description of the Archie tool is provided in our prior work [18], [31].

5.1 Preparation

To prepare *Archie* for use we have to install a library of tTPs and or create customized ones. For the purpose of our HADOOP case study we installed the library of ten tTPs described in this paper. However, tTPs can be created as needed to capture either traditional or customized architectural tactics. For example, an architect might design a customized solution which he deems to be architecturally significant. He then creates a customized tTP for the solution. To support automated tactic detection a tTP must be associated with a trained classifier; however, if this is not available, code can still be mapped manually. In the case that the architect creates his own customized tTP he will therefore need to map relevant code directly to the roles in the tTP. Once mapped, other tasks such as monitoring, visualizing, and notifying users of underlying architectural design concepts, can be supported in an automated manner.

5.2 Detecting Architectural Tactics

This paper has addressed the task of training a classifier to detect architectural tactics. As a result of this work we integrated the Tactic Detector into our *Archie* tool. The Tactic Detector not only performed well in both the n-fold and Hadoop experiments, but it has the added benefit of being self-explanatory i.e. it produces a list of weighted indicator terms which fully explain why a file is classified as a certain tactic or not. For example, in Figure 3 the displayed code relates to *heart beat* and relevant terms such as *beat* are highlighted. This makes it easier for the user to evaluate whether the identified tactic is correctly classified.

Archie allows a user to run, or re-run, the tactic detector against source code in order to generate a list of candidate architectural tactics. Because we cannot guarantee 100% precision in our results, we present the candidate tactics to the user for validation. The user must then confirm or deny each of the detected code snippets. This step can be performed immediately after running the tactic detector, later when the system notifies the user that they are editing architecturally significant code, or anytime in between.

5.3 Registering tactics with Event-Based Monitor

Archie implements an Event-based Traceability (EBT) infrastructure [18], [43] which allows all tactic-related classes to be mapped to specific roles in an instance of a tTP.

In our prior work we experimented with role-level classification of classes by training the tactic-detector to recognize specific roles such as heartbeat emitter and heartbeat receiver. We augmented the approach with light-weight structural analysis which took into account generalization and message passing relationships; however as reported in our prior work [20],

TABLE 9
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

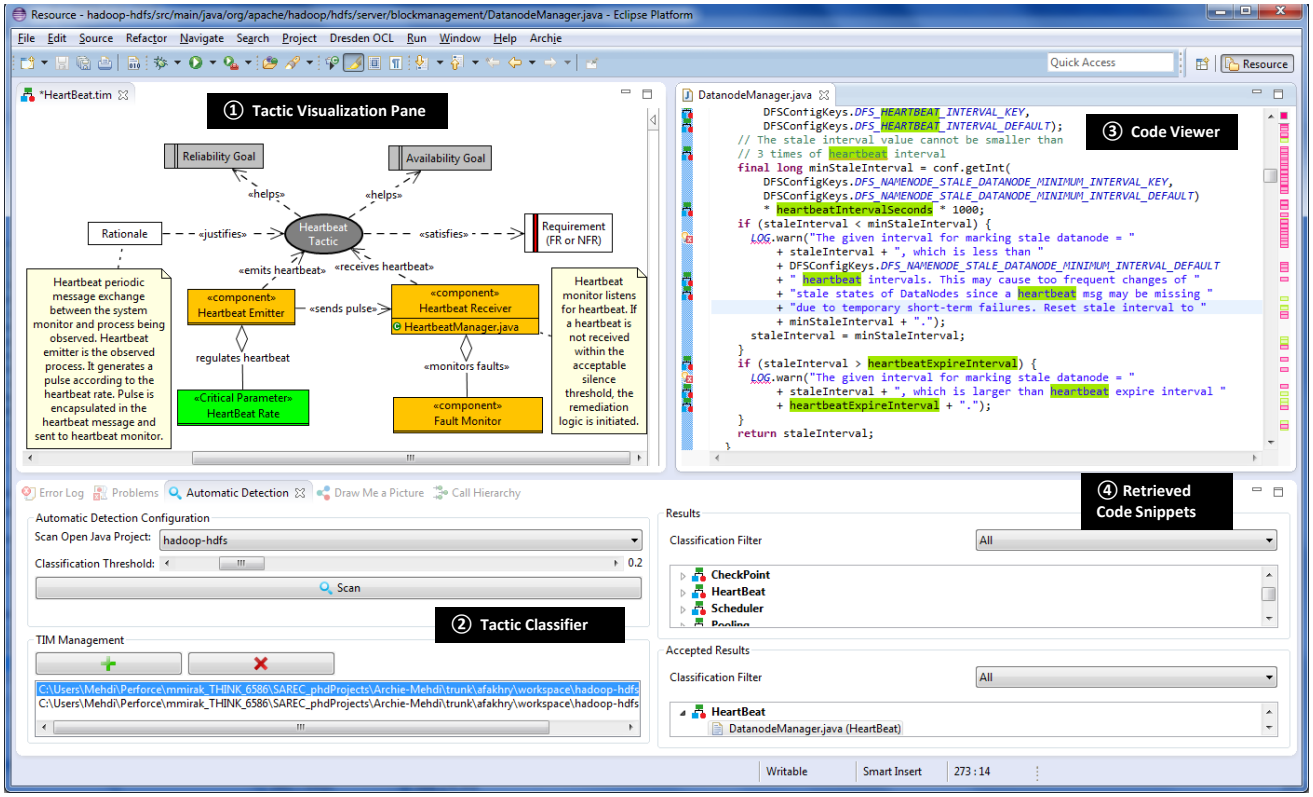

Fig. 3. Archie: Eclipse Plugin detects architectural tactics, monitors related code, and notifies developers when they modify architecturally significant parts of the code.

term differentiation across roles of a tactic provided inaccurate classification results. We therefore currently do not implement role-level classification in our *Archie* tool. Instead, a user is presented with a list of tactic-related classes. If validated by the user, these classes are mapped (i.e. registered) to the tactic; however, the user also has the option of mapping a class to a specific role in the tactic. The benefit of mapping at the role-level is that future notification messages and their related visualizations will be more informative.

It is necessary to create a unique instance of the relevant tTP for each observed instance of a tactic. For example, we found two cases of *Heartbeat* in Hadoop and therefore instantiated two different tTP instances, one of which is depicted in Figure 4. In this example, the *DataNode* class plays the role of heartbeat emitter, the *NameNode* class as receiver, and the *FSNameSystem* is responsible for monitoring the current state of heartbeats received by *NameNode*. These classes are mapped to their associated roles. Additional classes, such as *Balancer* and *BlockManager* interact closely with these ones to provide full heartbeat functionality and are mapped at a higher level of granularity to the overall tactic. It is the responsibility of the architect to decide whether to map classes to specific roles. The benefit of doing so is improved notification messages at runtime. Together, all of the mappings create traceability from the code via the *Heartbeat* tTP to goals and requirements related to *availability* and *reliability*.

We have mapped all of the Hadoop tactics depicted in Table 4 successfully to tTPs modeled in our Archie tool.

5.4 Monitoring

Once code is mapped to the tTP then attempts by developers to modify it will result in the generation of a notification message. An important, yet often unexplored research question addresses the issue of whether automatically reconstructed traceability links are good enough for use.

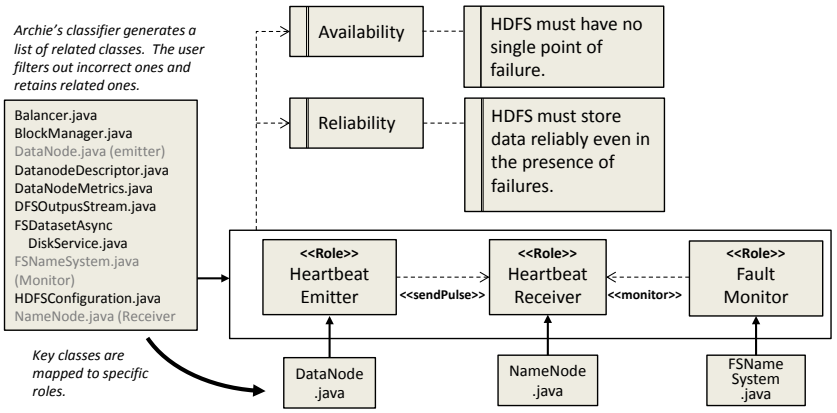

Fig. 4. In this instance of the heartbeat tactic found in the Hadoop Distributed File System (HDFS), DataNode.java, NameNode.java, and DSNameSystem.java were mapped respectively to emitter, receiver, and fault monitor roles.

We therefore designed a simulation which provides an initial evaluation of the usefulness of the generated coarse-grained traceability links for supporting software maintenance. We leave a complete user study for future work. The simulation was designed to reverse engineer the notifications that would have been generated by Archie had it been used over four releases of Hadoop, and to analyze the utility of these notification messages.

First, we used Archie to identify candidate architectural tactics for *Audit*, *Heartbeat*, *Scheduling*, *Resource Pooling*, and *Authentication* in version 0.20.1 of Hadoop. All of the automatically generated class-to-tTP trace links were used for this experiment, with no pre-filtering step to remove incorrect links. The experiment evaluated whether this approach resulted in a reasonable or unacceptable level of notification messages.

We mined the commit logs of Hadoop for versions 0.20.1 to 0.21.0 and for each release extracted a list of classes which had been modified. This list was compared against the tactic-related classes which had been mapped to tTPs in Archie. For each mapped class, we simulated the generation of a message to inform the developer about the underlying architectural tactic. For example, a modification made to the *HeartbeatManager.java* class resulted in a notification message stating that *'DatanodeManager.java' has been modified. This class plays a role in the heart beat tactic.* Each of the generated notifications were then evaluated by team members to determine whether the notification was correct or not. The notification was deemed correct if the class played a clear role in the tactic.

Table 10(a) reports on the number of correctly generated notifications (true positives), the number of unnecessary notifications (false positives), the number of missed notifications (false negatives), and the number of modifications which were correctly ignored (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 0.93 in all cases except for the *scheduling* tactic; however precision ranged from 0.35 to 0.96.

We then evaluated a second scenario in which we assumed that each incorrect notification would only have been sent once. This optimistically assumed that if an incorrect notification were received, the developer tagged it as incorrect, effectively rejecting the underlying mapping of class to tTP and leading to the removal of the link. Results from this simulation are reported in Table 10(b). Under these circumstances, user feedback from initial notifications significantly increased recall, precision, and specificity for all five tactics. In fact all metrics were over 0.92 except the precision for the *scheduling* tactic which remained at 0.73.

The results reported for this case study demonstrate that our approach generates tactic-grained links capable of supporting a practical task such as architectural preservation; 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 to develop a relatively accurate set of traces.

6 Threats to Validity

Threats to validity can be classified as *construct*, *internal*, *external*, and *statistical* validity. We discuss the threats which potentially impacted our work, and the ways in which we attempted to mitigate them.

External validity evaluates the generalizability of the approach. The primary threat is related to the construction of the datasets for this study. The first dataset included over five hundred samples of tactic-related code. The task of locating and retrieving

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2015.2479217, IEEE Transactions on Software Engineering

TABLE 10
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

these code snippets was conducted primarily by two members of our research team and was 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 identified code snippets was indeed representative of its relevant tactic. However, it is more difficult to ensure that all tactics of a given type have been identified. For example, there could be instances of tactics that we failed to find, despite the multi-faceted approach described in Section 3.2.1. Such tactics would likely be outliers and would represent ways of encoding a tactic that our approach was not trained to recognize, and which could be more difficult to classify than standard implementations. This issue can only be addressed over time.

In addition, all of the experiments conducted in our study were based on Java code. Some of the identified keyterms are influenced by java constructs such as calls to APIs that support specific tactic implementation. On the other hand, the majority of identified keyterms are non-java specific. We anticipate that retraining our approach on snippets from other languages would produce similar results to those reported in this paper.

Construct validity evaluates the degree to which the claims were correctly measured. The n-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 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.

Internal validity reflects the extent to which a study minimizes systematic error or bias, so that a causal conclusion can be drawn. 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 searching, browsing, and expert opinion. In the case of the Hadoop project, we elicited feedback from Hadoop developers on the open discussion forum.

Statistical validity concerns whether the statistical analysis has been conducted correctly. In order to address this threat appropriate statistical techniques were used. For reliability of conclusions we used both non-parametric and parametric tests. Uniformly both tests indicated that the Tactic Detector is more accurate than SVM, Slipper, AdaBoost and the Bayesian classification technique and this conclusion is statistically significant with a p-value of 0.05. While these results are taken from a single case study, they are supported by additional data points taken from over 50 high performance software systems.

7 Related Work

Related work falls primarily under three areas of documenting design rationales, reconstructing architectural knowledge, and automated traceability.

7.1 Documenting Design Rationales

As previously discussed in the introduction, several researchers have developed tools and techniques for documenting design rationales. For example, Burge et. al. developed a tool named Seurat [8]. Kruchten [9] developed a design rationale model that captured architectural design decisions, rationales, alternatives, and tradeoffs [9]. Other related approaches include the Architecture Design Decision Support System (ADDSS) [10], Process based Architecture Knowledge Management Environment (PAKME) [11], Architecture Rationale and Element Linkage (AREL) [12], and techniques for capturing and tracing architectural knowledge [44], [45].

However, in practice the cost and effort of documenting design rationales has been shown to be rather onerous [46] and therefore Gorton proposed focusing only on documenting architectural decisions when there is a clear payback in terms of criticality and/or usage of the documents [47]. Falessi et al., conducted two controlled industrial studies which confirmed the benefits of targeted documentation [48]. Our tTP-centric approach follows this recommended practice by documenting only critical architectural decisions. It

also alleviates cost and effort by providing automated support for detecting the tactics.

Existing design rationale solutions tend to focus on the design and fail to provide traceability down to the code level in which the decisions have been implemented [13]. As such, developers who wish to understand underlying architectural decisions must proactively seek out design information. In contrast our approach pushes relevant architectural knowledge to the developers as they modify the code. Furthermore, design rational approaches are manual in nature and generally fail to provide guidance on how to create and manage the potentially large number of traceability links that are needed to make the design knowledge available to a wide group of project stakeholders such as, architects, developers, and testers. They therefore provide little support for actually utilizing this knowledge during the software maintenance process.

7.2 Reconstructing Architectural Knowledge

Because design rationales and architectures are largely undocumented in many projects, researchers have developed techniques for reconstructing architectural knowledge [49]. The primary purpose of architecture reconstruction is to shine light on the program structure and design so that developers can either restore the intended architecture or rejuvenate it into a new optimal design. Reconstruction methods can be categorized as bottom-up, top-down, or hybrid [49], [50].

In bottom-up approaches low-level knowledge is mined and the abstraction level is progressively raised until a high level understanding of the architecture is achieved. [14, 153]. For example, Mancoridis et al., [51] created a tool named Bunch which leverages dependencies between classes to generate a call graph. This is used to cluster classes and create a high level structural abstraction of the system. Similarly Lungu et.al. developed a tool which utilizes dependencies between source files to generate a package view as well as high level modular view of the system [52]. Cai et al. utilized the notion of design rules to identify clusters of classes and infer the architecture from these clusters [53]. Maqbool et al. used hierarchical clustering to identify architectural structures [54]. Kazman et al. developed the Dali workbench which also supports bottom up reconstruction. Diverse samples of low-level data are collected and then relationships (i.e. tuples of "relation <entity1><entity2>") are identified. The architecture is then reverse engineered through a series of visualization and query steps [55].

Top-down approaches leverage architectural styles, patterns, requirements and other forms of high-level knowledge in an effort to match them to the source code. For example Murphy et al's reflexion model maps a hypothetical model of the intended architecture to the results from a static analysis of the source code. [56]. The Architecture Reconstruction Method (ARM) [57] searches for a set of user defined architectural patterns in order to reverse-engineer the implemented architecture. The user provides the pattern and also validates the retrieved instances. ARM uses the Rigi visualization tool to visualize the reconstructed architecture. Sartipi et al., [58] developed a similar approach based on pattern matching however they match patterns at the level of the abstract graph constructed from the source code.

Architectural reconstruction techniques do not solve the problem we attempt to address for several reasons. First, they typically reverse engineer high-level structural views of the system and fail to expose underlying design decisions. Second, they are effort intensive and often only partially automated. Finally, they do not provide traceability all the way from code to requirements and thereby have limited utility when it comes to pushing comprehensive information to developers during their maintenance tasks [49], [59].

In contrast, Jansen et al emphasize the recovery of design decisions [60]. Their Architectural Design Decision Recovery Approach (ADDRA) is based on differences in architectural design across different versions of the system. They first recover detailed designs for various versions and use these to generate a series of architectural views. They then use the delta between versions to identify design decisions. However their approach is manual in nature and also fails to establish traceability to code.

There is also a significant body of work in the area of design pattern detection [61]–[64]; however as previously explained, design pattern detection differs from the problem of tactic detection because a single tactic can often be implemented in numerous different ways and so we cannot rely upon structural analysis techniques.

Our approach could be classified as top-down reconstruction of architectural patterns. It is semi-automated in that it is capable of searching for and retrieving a fairly diverse set of patterns for which it has been trained. Given a tTP our approach also supports code to requirements traceability which means it has the ability to push useful information to developers.

7.3 Manual Architecture to Code Conformance

Researchers have developed techniques to connect architecture to source code in order to perform conformance checking [65]–[67]. However this body of work primarily focuses on architectural components as the major building blocks of the system, and traces these components to source code. For example ArchJava [66] uses the formal notion of Architecture Description Languages (ADL) and integrates architectural concepts into an implementation language (Java). This enables developers to present architectural features such as components, connectors and ports in their code and

to enforce communication integrity between code and design. In a similar approach, 1.x-way mapping maps architectural models (represented in xADL) to source code [67].

In contrast, instead of focusing on explicit structural properties of an architecture, our approach emphasizes architectural design decisions and their direct impact upon software qualities such as performance, reliability and availability.

7.4 Automated Traceability

Finally, our work is a special case of automated trace retrieval. Many researchers have investigated the use of basic search-based information retrieval methods to generate traces between documentation (i.e. requirements) and source code. Techniques have included probabilistic models [29], [68], the Vector Space Model (VSM) [69], LSI [70], and LDA [71], [72]. We utilized search-based approaches as one step in developing our training sets. However, such approaches can suffer from low recall when source and target artifacts use different terminology. As it is often the case that implemented tactics use code-oriented terminology (e.g. *isonlin* for the heartbeat tactic). Tactic-retrieval based purely on search is therefore unlikely to be consistently effective.

Another class of traceability solutions leverages the fact that certain types of software artifacts occur repeatedly across different software projects. This creates the opportunity to train a classifier to generate trace links. Such approaches have been shown to outperform simple search based approaches. Cleland-Huang et al., developed the NFR (non-functional requirement) classifier and used it to identify quality concerns such as security, performance, and usability which recur across requirements specifications in multiple projects [32]. Similarly Gibiec et al., used the same algorithm to trace regulatory standards to requirements [73], [74]. Architectural tactics also tend to recur across projects and therefore we utilized a similar training-based approach.

8 Conclusion

In this paper we have described our approach for automating the detection of architectural tactics in code and then leveraging the results to establish traceability from code to requirements. We have presented ten different tTPs, each one modeled to capture specific roles we observed across instances of the tactic in several different systems. We then trained seven different classifiers using code snippets extracted from 50 open source systems. When used to classify known tactic instances in the HDFS, the tactic detector outperformed other classifiers and retrieved most tactic types at recall rates of greater than 70 %.

The ultimate purpose of our work is to address the architectural erosion problem. While it was outside the scope of this paper to perform a longitudinal study in which our approach was used over time in a real project and demonstrated to reduce architectural erosion, we have successfully shown that Archie and the underlying classifiers have the ability to push architectural knowledge to developers as they make code changes within an IDE. We have further shown through the study of Hadoop's maintenance events that the quantity of notification messages is reasonable if user feedback filters out incorrectly classified classes.

In future work we plan to extend our library of tTPs to incoporate a broader range of architectural decisions. We will also evaluate our approach in a broader set of projects. Our Archie tool is released for public use at GitHub. (https://github.com/ArchieProject/Archie-Smart-IDE).

Acknowledgments

The work in this paper was partially funded US National Science Foundation grant # CCF-0810924 and Research Experience for Undergraduates (REU) grant # CCF 1341072. We particularly thank DePaul students Ahmed Fakhry, Artem Grechko, and Matteusz Wieloch for their efforts to help develop the Archie tool.

References

[1] H. Cervantes, P. V. Elizondo, and R. Kazman, "A principled way to use frameworks in architecture design." *IEEE Software*, vol. 30, no. 2, pp. 46–53, 2013. [Online]. Available: http://dblp.uni-trier.de/db/journals/software/software30.html#CervantesEK13

[2] R. K. Len Bass, Paul Clements, *Software Architecture in Practice*, 2000.

[3] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, pp. 40–52, October 1992. [Online]. Available: http://doi.acm.org/10.1145/141874.141884

[4] D. L. Parnas, "Software aging," in *Proceedings of the 16th international conference on Software engineering*, ser. ICSE '94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 279–287. [Online]. Available: http://dl.acm.org/citation.cfm?id=257734.257788

[5] G. Huang, H. Mei, and F.-Q. Yang, "Runtime recovery and manipulation of software architecture of component-based systems," *Automated Software Engg.*, vol. 13, no. 2, pp. 257–281, Apr. 2006. [Online]. Available: http://dx.doi.org/10.1007/s10515-006-7738-4

[6] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "A survey of software aging and rejuvenation studies," *J. Emerg. Technol. Comput. Syst.*, vol. 10, no. 1, pp. 8:1–8:34, Jan. 2014. [Online]. Available: http://doi.acm.org/10.1145/2539117

[7] 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.*, vol. 17, pp. 277–306, July 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1077863.1077864

[8] J. E. Burge and D. C. Brown, "Software engineering using rationale," *Journal of Systems and Software*, vol. 81, no. 3, pp. 395–413, 2008.

[9] P. Kruchten, "An ontology of architectural design decisions," *Groningen Workshop on Software Variability management*, pp. 55–62, 2004.

[10] R. Capilla, F. Nava, S. Pérez, and J. C. Dueñas, "A web-based tool for managing architectural design decisions," *SIGSOFT Softw. Eng. Notes*, vol. 31, Sept. 2006. [Online]. Available: http://doi.acm.org/10.1145/1163514.1178644

[11] M. A. Babar and I. Gorton, "A tool for managing software architecture knowledge," in *Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent*, ser. SHARK-ADI '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 11–. [Online]. Available: http://dx.doi.org/10.1109/SHARK-ADI.2007.1

[12] A. Tang, Y. Jin, and J. Han, "A rationale-based architecture model for design traceability and reasoning," *Journal of Systems and Software*, vol. 80, no. 6, pp. 918 – 934, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0N-4M6SB7T-1/2/82ea7eabc2c4947aee2168fd536cc9f3

[13] M. Mirakhorli and J. Cleland-Huang, "Tracing architectural concerns in high assurance systems: (nier track)," in *ICSE*, 2011, pp. 908–911.

[14] A. Jansen and J. Bosch, "Software architecture as a set of architectural design decisions," in *WICSA*, 2005, pp. 109–120.

[15] P. Kruchten, R. Capilla, and J. C. Dueas, "The decision view's role in software architecture practice," *IEEE Software*, vol. 26, no. 2, pp. 36–42, 2009.

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

[17] M. Mirakhorli and J. Cleland-Huang, "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*, ser. ICSM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 123–132. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2011.6080779

[18] M. Mirakhorli, A. Fakhry, A. Grechko, M. Wieloch, and J. Cleland-Huang, "Archie: a tool for detecting, monitoring, and preserving architecturally significant code," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 739–742. [Online]. Available: http://doi.acm.org/10.1145/2635868.2661671

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

[20] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar, "A tactic centric approach for automating traceability of quality concerns," in *International Conference on Software Engineering, ICSE (1)*, 2012.

[21] D. P. Siewiorek and P. Narasimhan, "Fault-tolerant architectures for space and avionics applications," NASA Ames Research http://ic.arc.nasa.gov/projects/ishem/Papers/Siewi.

[22] J. Aplin, "Primary flight computers for the boeing 777," *Microprocessors and Microsystems*, vol. 20, no. 8, pp. 473–478, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0X-4GW41CM-4/2/15ff064161dbc81c71c28316605e931d

[23] NASA's and Robots, *Online at: http://prime.jsc.nasa.gov/ ROV/n-links.html*, 2008.

[24] J. Hart, E. King, P. Miotto, and S. Lim, *Orion GN&C Architecture for Increased Spacecraft Automation and Autonomy Capabilities*, AIAA, August 2008.

[25] S. Tamblyn, H. Hinkel, and D. Saley, *Crew Exploration Vehicle (CEV) Reference Guidance, Navigation, and Control (GN&C) Architecture*, 30th ANNUAL AAS GUIDANCE AND CONTROL CONFERENCE, February 2007.

[26] ——, *NASA Exploration Systems Architecture Study (ESAS) Final Report*, 2005. [Online]. Available: http://www.nasa.gov/pdf/140649main_ESAS_full.pdf

[27] 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.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[29] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, no. 10, pp. 970–983, Oct. 2002. [Online]. Available: http://dx.doi.org/10.1109/TSE.2002.1041053

[30] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova, "Best practices for automated traceability," *Computer*, vol. 40, no. 6, pp. 27–35, 2007.

[31] M. Mirakhorli, "Preserving the quality of architectural decisions in source code, PhD Dissertation, DePaul University Library," 2014.

[32] J. Cleland-Huang, R. Settimi, X. Zou, and P. Solc, "Automated detection and classification of non-functional requirements," *Requir. Eng.*, vol. 12, no. 2, pp. 103–120, 2007.

[33] T. Joachims, "Text categorization with suport vector machines: Learning with many relevant features," in *Proceedings of the 10th European Conference on Machine Learning*, ser. ECML '98. London, UK, UK: Springer-Verlag, 1998, pp. 137–142. [Online]. Available: http://dl.acm.org/citation.cfm?id=645326.649721

[34] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins, "Text classification using string kernels," *J. Mach. Learn. Res.*, vol. 2, pp. 419–444, Mar. 2002. [Online]. Available: http://dx.doi.org/10.1162/153244302760200687

[35] S. Tong and D. Koller, "Support vector machine active learning with applications to text classification," *J. Mach. Learn. Res.*, vol. 2, pp. 45–66, Mar. 2002. [Online]. Available: http://dx.doi.org/10.1162/153244302760185243

[36] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[37] A. Genkin, D. D. Lewis, and D. Madigan, "Large-scale bayesian logistic regression for text categorization," *Technometrics*, vol. 49, pp. 291–304(14), August 2007. [Online]. Available: http://www.ingentaconnect.com/content/asa/tech/2007/00000049/00000003/art00007

[38] Y. Freund and R. E. Schapire, "Experiments with a new boosting algorithm," in *Thirteenth International Conference on Machine Learning*. San Francisco: Morgan Kaufmann, 1996, pp. 148–156.

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

[40] L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, Aug. 1996. [Online]. Available: http://dx.doi.org/10.1023/A:1018054314350

[41] ——, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, Aug. 1996. [Online]. Available: http://dx.doi.org/10.1023/A:1018054314350

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

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

[44] A. Jansen and J. Bosch, "Software architecture as a set of architectural design decisions," in *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 109–120. [Online]. Available: http://portal.acm.org/citation.cfm?id=1130239.1130657

[45] A. Jansen, J. Bosch, and P. Avgeriou, "Documenting after the fact: Recovering architectural design decisions," *Journal of Systems and Software*, vol. 81, no. 4, pp. 536 – 557, 2008, ¡ce:title¿Selected papers from the 10th Conference on Software Maintenance and Reengineering (CSMR 2006)¡/ce:title¿. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016412120700194X

[46] J. Lee, "Design rationale systems: understanding the issues," *IEEE Expert*, pp. 78–85, 1997.

[47] I. Gorton, *Essential Software Architecture*. Springer-Verlag New York, Inc., 2006.

[48] D. Falessi, L. C. Briand, G. Cantone, R. Capilla, and P. Kruchten, "The value of design rationale information," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, p. 21, 2013. [Online]. Available: http://doi.acm.org/10.1145/2491509.2491515

[49] R. Koschke, "Architecture reconstruction," in *Software Engineering, International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures*, 2008, pp. 140–173. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-95888-8_6

[50] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Trans. Software*

Eng., vol. 35, no. 4, pp. 573–591, 2009. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2009.19

[51] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *Proceedings of the IEEE International Conference on Software Maintenance*, ser. ICSM '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 50–. [Online]. Available: http://dl.acm.org/citation.cfm?id=519621.853406

[52] M. Lungu, M. Lanza, and O. Nierstrasz, "Evolutionary and collaborative software architecture recovery with softwarenaut," *Science of Computer Programming*, vol. 79, pp. 204–223, 2014.

[53] Y. Cai, H. Wang, S. Wong, and L. Wang, "Leveraging design rules to improve software architecture recovery," in *Proceedings of the 9th international ACM SIGSOFT conference on Quality of Software Architectures, QoSA 2013, part of Comparch '13 Federated Events on Component-Based Software Engineering and Software Architecture, Vancouver, BC, Canada, June 17-21, 2013*, 2013, pp. 133–142. [Online]. Available: http://doi.acm.org/10.1145/2465478.2465480

[54] O. Maqbool and H. A. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Trans. Software Eng.*, vol. 33, no. 11, pp. 759–780, 2007. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2007.70732

[55] R. Kazman and S. J. Carrière, "Playing detective: Reconstructing software architecture from available evidence," *Autom. Softw. Eng.*, vol. 6, no. 2, pp. 107–138, 1999. [Online]. Available: http://dx.doi.org/10.1023/A:1008781513258

[56] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: Bridging the gap between design and implementation," *IEEE Trans. Software Eng.*, vol. 27, no. 4, pp. 364–380, 2001. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/32.917525

[57] G. Y. Guo, J. M. Atlee, and R. Kazman, "A software architecture reconstruction method," in *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, ser. WICSA1. Deventer, The Netherlands, The Netherlands: Kluwer, B.V., 1999, pp. 15–34. [Online]. Available: http://dl.acm.org/citation.cfm?id=646545.696370

[58] K. Sartipi, "Software architecture recovery based on pattern matching," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 293–. [Online]. Available: http://dl.acm.org/citation.cfm?id=942800.943546

[59] A. Dragomir, M. F. Harun, and H. Lichter, "On bridging the gap between practice and vision for software architecture reconstruction and evolution: A toolbox perspective," in *Proceedings of the WICSA 2014 Companion Volume*, ser. WICSA '14 Companion. New York, NY, USA: ACM, 2014, pp. 10:1–10:4. [Online]. Available: http://doi.acm.org/10.1145/2578128.2578235

[60] A. Jansen, J. Bosch, and P. Avgeriou, "Documenting after the fact: Recovering architectural design decisions," *J. Syst. Softw.*, vol. 81, no. 4, pp. 536–557, Apr. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2007.08.025

[61] 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*, vol. 10, pp. 4: 1–31, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/jot/jot10.html#FontanaZM11

[62] N. Pettersson, W. Lwe, and J. Nivre, "Evaluation of accuracy in design pattern occurrence detection." 2010, pp. 575–590.

[63] G. Antoniol, G. Casazza, M. D. Penta, and R. Fiutem, "Object-oriented design patterns recovery," *Journal of Systems and Software*, vol. 59, no. 2, pp. 181–196, 2001. [Online]. Available: http://dx.doi.org/10.1016/S0164-1212(01)00061-9

[64] G. Rasool, P. Maeder, and I. Philippow, "Evaluation of design pattern recovery tools," *Procedia CS*, vol. 3, pp. 813–819, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.procs.2010.12.134

[65] J. Van Eyck, N. Boucké, A. Helleboogh, and T. Holvoet, "Using code analysis tools for architectural conformance checking," in *Proceedings of the 6th International Workshop on SHAring and Reusing Architectural Knowledge*, ser. SHARK '11. New York, NY, USA: ACM, 2011, pp. 53–54. [Online]. Available: http://doi.acm.org/10.1145/1988676.1988687

[66] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: Connecting software architecture to implementation," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 187–197. [Online]. Available: http://doi.acm.org/10.1145/581339.581365

[67] Y. Zheng and R. N. Taylor, "Enhancing architecture-implementation conformance with change management and support for behavioral mapping," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 628–638. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337297

[68] J. Cleland-Huang, R. Settimi, C. Duan, and X. Zou, "Utilizing supporting evidence to improve dynamic requirements traceability," in *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, ser. RE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 135–144. [Online]. Available: http://dx.doi.org/10.1109/RE.2005.78

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

[70] D. Poshyvanyk, M. Gethers, and A. Marcus, "Concept location using formal concept analysis and information retrieval," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 4, pp. 23:1–23:34, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2377656.2377660

[71] J. Hayes, A. Dekhtyar, and S. Sundaram, "Advancing candidate link generation for requirements tracing: the study of methods," *Software Engineering, IEEE Transactions on*, vol. 32, no. 1, pp. 4–19, Jan 2006.

[72] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, 2010, pp. 95–104. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806817

[73] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker, "A machine learning approach for tracing regulatory codes to product specific requirements," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, 2010, pp. 155–164. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806825

[74] M. Gibiec, A. Czauderna, and J. Cleland-Huang, "Towards mining replacement queries for hard-to-retrieve traces," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, 2010, pp. 245–254. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859046