Archie: A Tool for Detecting, Monitoring, and Preserving Architecturally Significant Code

Mehdi Mirakhorli
Software Engineering Department
Rochester Institute of Technology
Rochester, NY USA
mehdi@se.rit.edu

Ahmed Fakhry, Artem Grechko,
Mateusz Wieloch, Jane Cleland-Huang
School of Computing
DePaul University
Chicago, IL. USA
jhuang@cs.depaul.edu

ABSTRACT

The quality of a software architecture is largely dependent upon the underlying architectural decisions at the framework, tactic, and pattern levels. Decisions to adopt certain solutions determine the extent to which desired qualities such as security, availability, and performance are achieved in the delivered system. In this tool demo, we present our Eclipse plug-in named *Archie* as a solution for maintaining architectural qualities in the design and code despite long-term maintenance and evolution activities. Archie detects architectural tactics such as heartbeat, resource pooling, and role-based access control (RBAC) in the source code of a project; constructs traceability links between the tactics, design models, rationales and source code; and then uses these to monitor the environment for architecturally significant changes and to keep developers informed of underlying design decisions and their associated rationales.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: Object Oriented Design Methods; D.2.1 [**Software Architectures**]: Patterns

General Terms

Design, Performance, Reliability

Keywords

Architecture, Degradation, Patterns, Tactics

1. INTRODUCTION

Architectural knowledge depicting design decisions and their associated rationales, is often undocumented and tacit in nature. Given the size, complexity, and longevity of many projects, developers can lose track of early design decisions, and new developers may fail to acquire a comprehensive understanding of them. As a result, design qualities tend to erode during refactoring, bug fixing, and other kinds of maintenance activities [14, 13]. Fortunately, such degradation can be partially prevented by exposing underlying design decisions, architectural tactics, styles and constraints to software developers as they plan and implement changes [2]. Existing tools [7] attempt to achieve this goal by capturing rationales at the design level; however they typically fail to connect architectural decisions to the implemented code. As a result, developers initiating changes at the code level often fail to fully understand the underlying design decisions.

An alternate solution is to use trace links to connect design rationales with impacted parts of the code [11]. Unfortunately, while it is conceptually simple to create a trace matrix documenting the relationships between design decisions, their rationales, and the impacted code elements, in practice any efforts to establish traceability at the code level are very challenging [12]. There are several contributing factors including lack of adequate tooling and the fact that traceability links are often created, maintained, and used in isolation from regular development activities, and are not accessible to support daily software engineering tasks [6].

To address these problems we present an eclipse plugin named *Archie*. Archie is designed to help automate the creation and maintenance of architecturally-relevant trace links between code, architectural decisions, and related requirements. Once created, the links are then used to actively monitor architecturally significant areas of the code, and to generate timely user-notifications describing underlying architectural decisions. In this way, Archie ensures that developers working in sensitive areas of the code, are fully informed of the impact of modifications and refactorings – thereby contributing to preserving architectural qualities and mitigating the long-term problem of architectural decay in the system. These features build upon concepts described in our prior work [8, 10, 12].

Archie is released as an open source project in GitHub under the name Archie-Smart-IDE [1]. A streamlined version of it is also released in the Software Assurance Marketplace (SWAMP) [5]. Furthermore we provide an introductory demo describing Archie's main features at `http://re.cs.depaul.edu/mehdi/Archie.mp4`. In the remainder of this paper we present Archie's features providing illustrations taken from the HADOOP-HDFS system.

2. OVERVIEW OF FEATURES

To support our goal of architectural preservation Archie includes various capabilities, several of which are depicted

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2661671

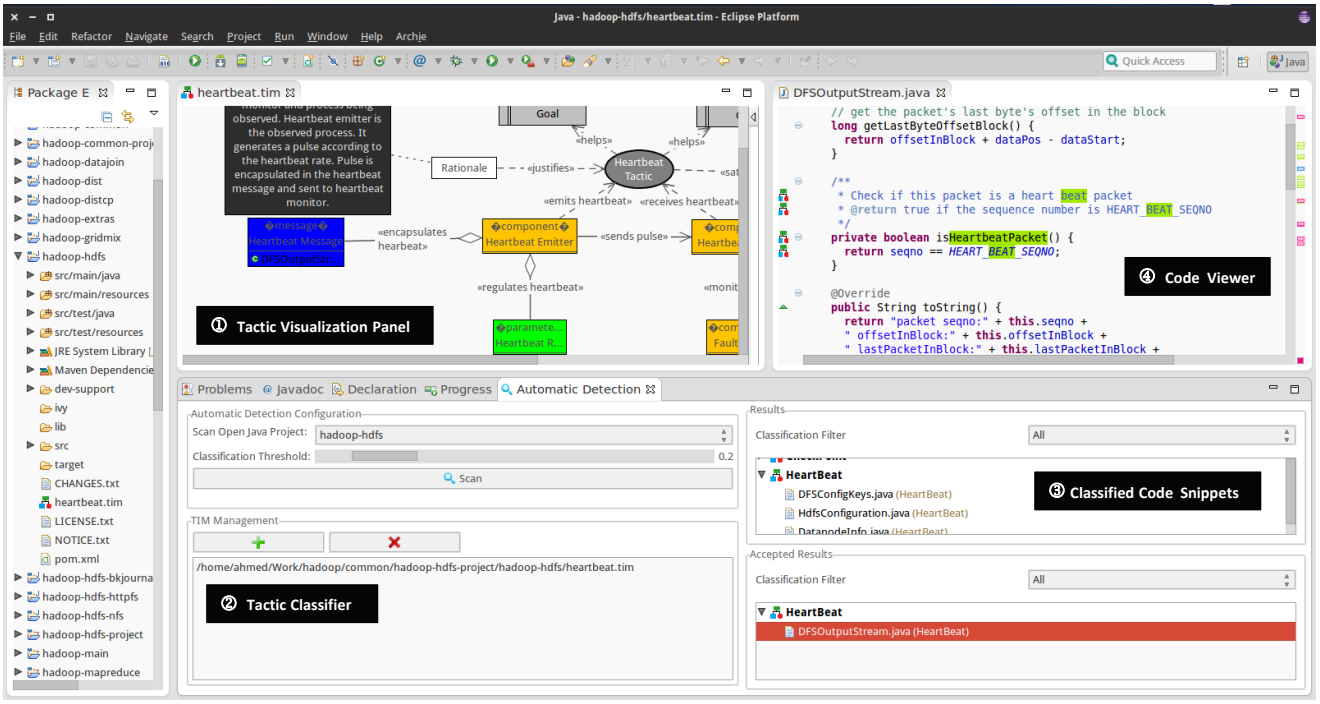

Figure 1: Archie: Eclipse Plugin that detects architectural tactics, monitors related code, and notifies developers when they modify architecturally significant parts of the code.

in Figure 1. These include:

- A **detection engine** capable of identifying sections of the code which implement architectural decisions which Archie's classifiers have been trained to detect, and an interactive viewer which allows a user to browse through code snippets returned by the detection engine.
- An **annotated code viewer**, which highlights architecturally significant parts of the code.
- **Visualization** features for (1) generating views of specific architectural tactics, their relationships to design rationales and requirements, and (2) generating global views of architectural decisions.
- Features to allow a user to bypass the automated detection process, and **manually mark-up sections of code** as being architecturally significant.
- 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.

2.1 Tactic Traceability Patterns

Archie is built around the fundamental concept of a Tactic Traceability Pattern (TTP). We first introduce this concept, and then discuss the related functionality. A TTP (referred to as a tactic Traceability Information Model in our prior work [8, 10]), captures the primary roles of an architectural tactic. For example, the primary roles of the *heartbeat* tactic include *emitter*, *receiver*, and *health monitor*. A TTP captures the relationships between these roles i.e. an emitter component sends messages to a receiver, while a health monitor takes actions if the monitored component fails. Furthermore, a TTP also captures the underlying rationales for using the tactic. These typically come in the form of a description that explains the quality concern being addressed. Each of the provided TTPs is initially populated with a set of default rationales. For example, in the case of heartbeat, these are related to reliability of a critical component. A user can modify rationales and also add references to relevant requirements. Archie ships with a basic set of TTPs including heartbeat, audit, authorization, resource pooling, and scheduler; however a user can utilize Archie's drag-and-drop modeling features to create customized TTPs.

TTPs provide three primary benefits. First, they reduce the cost and effort of establishing and maintaining trace links between architectural tactics and code by providing traceability guidance to the user [10]. Secondly, TTPs reduce the tracing process to a simple mapping task. Instead of documenting trace links in an external trace matrix, the TTP allows a user to establish a trace link by selecting a segment of code and using mouse clicks to map it to either the entire TTP or to a specific role. Finally, the TTP provides a visual framework for communicating underlying architectural knowledge to the users.

2.2 Architectural Code Detection

Archie includes a set of code-based classifiers constructed to detect different architectural tactics in the source code [9, 12]. The underlying classification algorithm is a modified version of our previously developed approach for classifying textual entities such as non-functional requirements and regulatory codes [4]. Archie's individual classifiers have been trained to detect audit, asynchronous method invocation, authentication, checkpoint, heartbeat, role-based access control (RBAC), resource pooling, scheduler, and secure session tactics. The classifiers were trained using code snippets of different architectural tactics collected from hun-

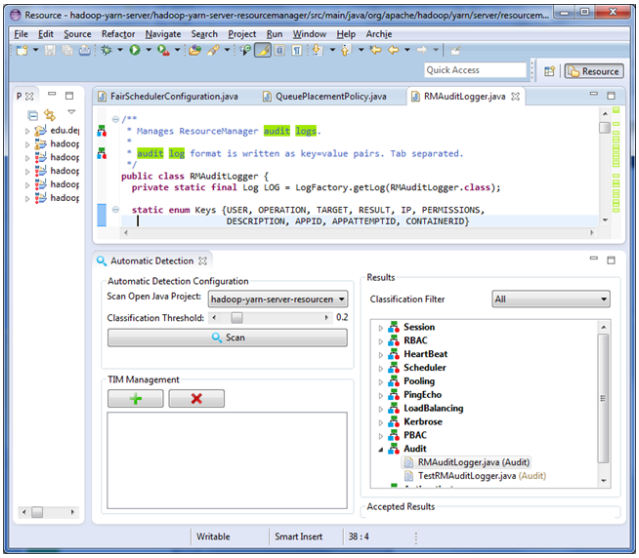

Figure 2: Archie's Tactic Detection features

dreds of high-performance, open source projects. During this phase, the classifier learns the terms that developers typically use to implement each architectural tactic and assigns each potential indicator term a weight with respect to each type of architectural tactic. The weight estimates how strongly an indicator term represents an architectural tactic. For instance, the term *priority* is found more commonly in code related to the *scheduling* tactic than in other kinds of code, and therefore the classifier assigns it a higher weighting with respect to scheduling. In the classification phase, these indicator terms are used to identify sections of the code which are tactic-related. Accuracy metrics, derived from Hadoop project [9], are reported in Table 1, and show that F-Measure (i.e. the harmonic mean of recall and precision) returned scores of 0.80 or higher for five tactics, 0.75 or higher for an additional two, and 0.48, and 0.66 for the remaining two. A complete description of the classification algorithm and its evaluation against an extensive set of architectural tactics can be found at [9]. Our current version of Archie has pretrained classifiers for each of these tactics.

Table 1: Accuracy of Trained Classifier applied to Tactics in the HADOOP-HDFS System

Tactic	Recall	Precision	F-Measure
Audit	0.71	1.00	0.83
Asynch. Invoc.	0.72	1.00	0.84
Authentication	0.70	0.61	0.66
Checkpoint	1.00	1.00	1.00
Heartbeat	1.00	0.66	0.79
RBAC	0.97	0.31	0.48
Resource Pooling	1.00	0.88	0.93
Scheduler	0.94	0.65	0.77
Secure Session	0.84	0.84	0.84

Classifiers are launched against the code in a currently open Eclipse project using Archie's features depicted in Figure 2. The set of identified code files categorized by tactic are then displayed to the user. The user must inspect each of the returned files and validate that they are tactic-related. Alternately, the user could accept all returned files for a

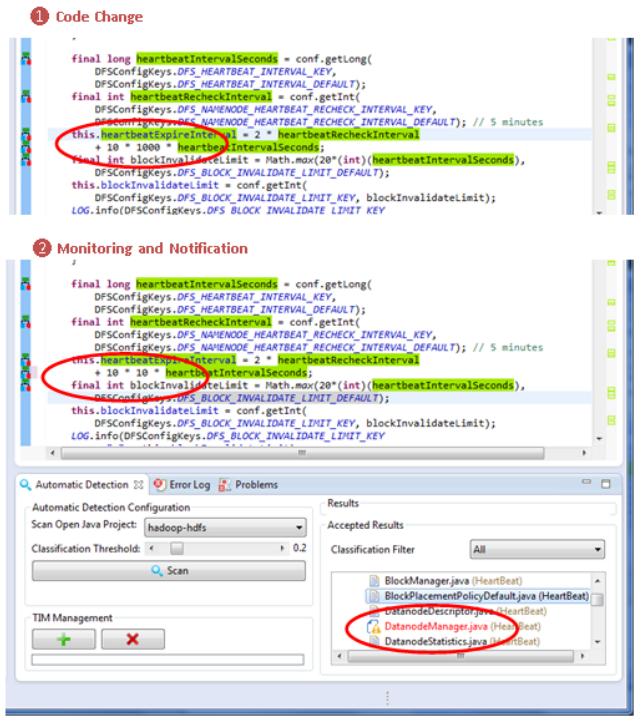

Figure 3: An architecture protection scenario

specific tactic type and defer the review process until later when they are actually used to generate notifications. Once a set of files have been identified for a given architectural tactic, they need to be mapped to instances of the TTP. For each identified instance of a tactic, the user instantiates a new TTP and by simply clicking on the various parts of the TTP, maps each relevant file to either a general instance of the tactic or to its specific roles – thereby establishing traceability from the code via the tactic to the underlying design rationales. As none of the classifiers are 100% accurate, it is likely that some relevant code will be missed by the classifier. However, *Archie* allows the user to bypass the classifier and to manually associate a file with a specified tactic.

2.3 Runtime Monitoring

All architecturally significant code which has been mapped to TTPs is monitored for change activity. *Archie* integrates an event-based traceability engine [3] in which all classes mapped to a TTP are registered with the event server, and the visualization panel is registered as a subscriber. Whenever a user modifies architecturally significant code (i.e. code mapped to a TTP) a notification event is triggered, and underlying architectural knowledge is visualized. For example, as depicted in Figure 3, a programmer views a class that implements heartbeat emitter functionality in Hadoop *dataNode*. The monitoring system highlights all tactic-related code, using a different color for each tactic. In this example, the *heartbeat*-related code is highlighted. At the same time, the heartbeat TTP is visualized showing the developer that code in *datanode.java* serves as the heartbeat emitter, and that it sends heartbeats to *HeartBeatManager*.

2.4 Visualization

In addition to visualizing specific tactics using the TTPs, Archie provides options for visualizing more general archi-

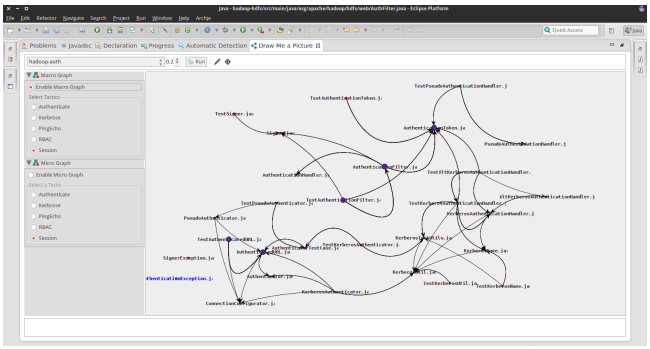

Figure 4: Macro-level view of the secure session manager within the context of other files in one Hadoop module

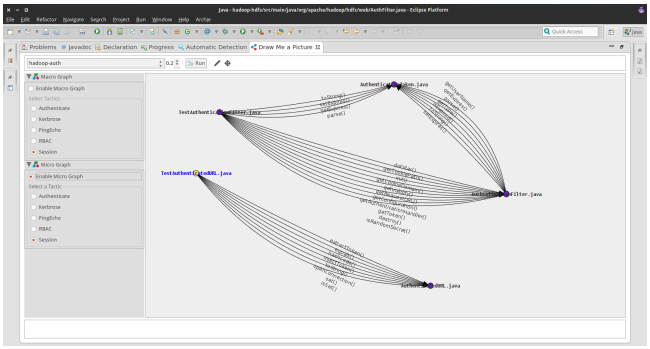

Figure 5: Micro-level view of the secure session manager showing only tactic related files

tectural knowledge. These visualizations are developed using the Java Universal Network/Graph Framework (JUNG) which is available on sourceforge. We depict two examples showing a micro (Figure 4), and macro (Figure 5) view of the secure-session management tactic as implemented in Hadoop's HDFS system. Such views provide additional insights into the underlying architectural decisions.

3. PERFORMANCE

Archie's classification algorithm is highly scalable. It has been tested on several systems ranging from 1,000 to 20,000 java files. On the Apache Hadoop HDFS system, which contains approximately 1,700 files, Archie classified 9 tactics in less than 30 seconds. On the other hand, training new classifiers can be time consuming; however this is performed off-line and does not affect runtime performance of Archie.

4. CONCLUSION

Archie includes additional features which we are unable to present here, such as a fully implemented trace retrieval engine for dynamically tracing code to external documentation. However, Archie's primary contribution is in the area of architectural preservation through detecting and tracing architectural concerns, and then using these trace links to keep developers fully informed of underlying architectural knowledge. In addition to the Eclipse plugin presented here, we have also prototyped similar functionality at the design level in Enterprise Architect. A screencast demo showing many of Archie's features is available at: http://re.cs.depaul.edu/mehdi/Archie.mp4.

5. ACKNOWLEDGMENTS

The work in this paper was partially funded by the US National Science Foundation grant # CCF-0810924 and by funding from the U.S. Department of Homeland Security in conjunction with the Security and Software Engineering Center (S^2ERC).

6. REFERENCES

[1] Archie-smart-ide. Available on GitHub at https://github.com/ArchieProject/Archie-Smart-IDE, June 2014.

[2] G. Booch. Draw me a picture. *IEEE Software*, 2011.

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

[4] 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.

[5] Continuous Software Assurance Marketplace. *https://continuousassurance.org/*, 2014.

[6] O. Gotel and A. Finkelstein. Contribution structures [requirements artifacts]. In *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on*, pages 100 – 107, Mar. 1995.

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

[8] M. Mirakhorli. Tracing architecturally significant requirements: a decision-centric approach. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1126–1127, New York, NY, USA, 2011. ACM.

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

[10] 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*, ICSM '11, pages 123–132, Washington, DC, USA, 2011. IEEE Computer Society.

[11] M. Mirakhorli and J. Cleland-Huang. Tracing non-functional requirements. In *Software and Systems Traceability*, pages 299–320. 2012.

[12] 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.

[13] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17:40–52, October 1992.

[14] 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.