Agile Architecture: Bringing Design Thinking into Developers Daily Activities.

Mehdi Mirakhorli and Daniel E. Krutz

Department of Software Engineering, Rochester Institute of Technology
Rochester, NY USA, 14606
{mehdi,dkrutz}@se.rit.edu

Abstract. Agile software development advocates iterative incremental development of architecture, where the architectural solutions are devised as new requirements are discovered. The quality of software is largely dependent upon the underlying architectural decisions at the framework, tactic, and pattern levels. Unfortunately, few tools are capable of fully supporting software architecture development which work well with agile development practices and culture. In this research paper, we present our Eclipse plug-in named *Archie* as a solution for integrating architecture activities into agile development environments. It supports developers during incremental architecture development and maintenance. Archie has features for automating design documentation and communication. It automatically detects architectural tactics such as heartbeat, resource pooling, and role-based access control (RBAC) in the source code of a project; and constructs traceability links between the tactics, design models, rationales and source code. These are then used to monitor the environment for architecturally significant changes and to keep developers informed of underlying design decisions and their associated rationales.

Key words: Agile Architecture, Tool, Tactical Spikes

1 Introduction

Agile software development is a method of managing software projects to make them more adaptable to change and to reduce costs associated with unnecessary documentation. Agile development emphasizes close collaboration between developers and business experts, frequent face to face communication, continual delivery of incremental working software, and self organizing teams. This is a stark contrast to development techniques such as Waterfall, that typically require significant amounts of up front requirements analysis, architecture design and continuous documentation. This often makes project changes more difficult and expensive.

The reduction of excessive up front design and unneeded formal architectural documentation increases the necessity for other methods of addressing quality, communication, and interactions between developers and stakeholders. In agile software development, architecture design is integrated with coding activities, advocating less up front design and more incremental architectural spikes to address each quality requirements. In such an environment, there is also less emphasize on documenting architectural knowledge from driving requirements to the adopted patterns, tactics and styles. The rationale behind these choices are maintained socially through practices such as pair programming, collective ownership and stand up meetings.

Currently, agile techniques are supported with tools which promote core agile practices. For example there are several mature tools for code refactoring and test driven design [1, 21, 23]. Unfortunately, there are few tools which assist developers in agile architecture development, design maintenance [9, 20] and documentation. Architecture centric tools [13, 14, 28] assume that an up front architecture design processes exist and heavy design artifacts are created. This makes them impractical in many iterative incremental projects with small cycles of design and often leads developers to not adopt these tools. Instead, they are often forced to rely upon social techniques to develop, communicate, and maintain their architecture.

The lack of architecture centric tools that fit the agile development paradigm and culture can increase the chances of quality degradation, where the implementation of architectural choices to satisfy quality concerns are drifted from the initial intends resulting design erosion and degradation of software qualities. In Robert Martin's recent book entitled "Agile Software Development - Principles, Patterns, and Practices" [15], he commented that while developers may initially release a system that meets the intended design, it does not take long before "the software starts to rot like a piece of bad meat" leading to problems such as rigidity, fragility, and unnecessary complexity of the design.

This problem is exacerbated by the fact that popular software engineering tools and environments fail to make underlying design decisions visible to programmers. This results in maintainers not being kept fully informed of the relevant underlying patterns, tactics, and constraints as they build, maintain, and refactor a software system [4].

In this paper, we introduce a pluggable tool *Archie*, which can be used to support different agile-architecture development activities. This tool was initially developed to support integrated architecture development and maintenance. However the automation features of Archie makes it suitable for agile projects. Archie has features for helping developers devise incremental architectural choices, proactively sharing design knowledge with programmers, and keeping them informed of underlying architectural decisions during coding activities. Archie helps developers to perform change impact analysis of architectural concerns at the code level, and provides infrastructure to enable the concept of "Design Ownership", supporting the developers to obtain accountability for their design choices.

2 A tool for Agile Architecture Development and Maintenance

To support our goal of integrating architecture thinking in coding activities, Archie delivers various capabilities, several of which are depicted in Figure 1. These include:

• A **quality driven design dashboard** which helps developers in finding appropriate design choices to address software qualities. Archie contains a catalogue of quality attributes and related tactics which can be used to implement and satisfy each quality concern.

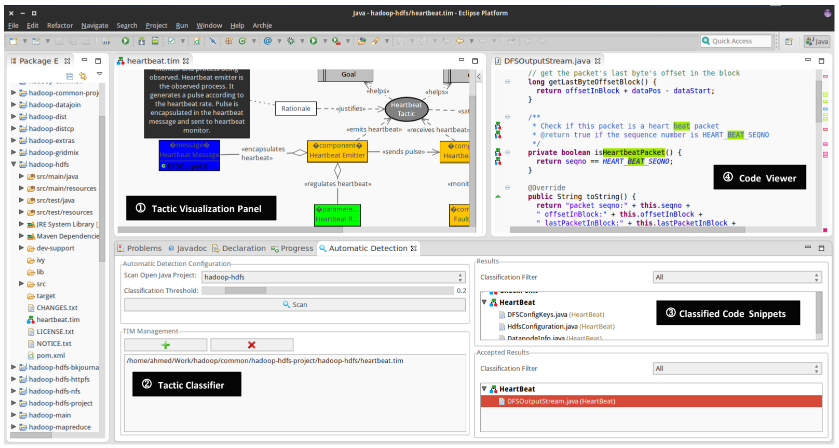
• An **engine to automatically detect tactical spikes.** Archie is capable of identifying sections of the code which implement architectural decisions, along with an interactive viewer which allows a user to browse through code snippets returned by the detection engine.

• An **annotated code viewer** which highlights architecturally significant parts of the code.

• **Visualization of tactical spikes** 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 **architecture ownership engine** which constantly monitors changes to the code in the background, notifies the owner of tactical spikes and the developer when they start to modify sensitive areas of the code, and displays information about the underlying architectural decisions.



observed. Heartbeat emitter is the observed process. It generates a pulse according to the heartbeat rate. Pulse is encapsulated in the heartbeat message and sent to heartbeat monitor.

```
// get the packet's last byte's offset in the block
long getLastByteOffsetBlock() {
  return offsetInBlock + dataPos - dataStart;
}

/**
 * Check if this packet is a heart beat packet
 * @return true if the sequence number is HEART_BEAT_SEQNO
 */
private boolean isHeartbeatPacket() {
  return seqno == HEART_BEAT_SEQNO;
}

@Override
public String toString() {
  return "packet seqno:" + this.seqno +
  " offsetInBlock:" + this.offsetInBlock +
  " lastPacketInBlock:" + this.lastPacketInBlock +
```

Automatic Detection Configuration
Scan Open Java Project: hadoop-hdfs
Classification Threshold: 0.2

TIM Management
/home/ahmed/Work/hadoop/common/hadoop-hdfs-project/hadoop-hdfs/heartbeat.tim

Results
Classification Filter: All
HeartBeat
DFSConfigKeys.java (HeartBeat)
HdfsConfiguration.java (HeartBeat)
DatanodeInfo.java (HeartBeat)

Accepted Results
Classification Filter: All
HeartBeat
DFSOutputStream.java (HeartBeat)

Fig. 1: A screen-shot of Archie, depicting different features and dashboards

3 Quality Driven Design: Tactical Spikes

Customer satisfaction in any software product is highly dependent on the extent to which quality concerns such as performance, security, reliability, and availability [12] are addressed. In practice, a rich set of proven and re-usable architectural solutions in terms of tactics can be used to satisfy each specific quality attribute [3]. Architectural tactics, come in many different shapes and sizes and describe solutions for a wide range of quality concerns [10]. For instance, reliability tactics provide solutions for fault mitigation, detection, and recovery. Performance tactics grant solutions for resource contention to optimize response time and throughput. Security tactics provide solutions for authorization, authentication, non-repudiation and other similar factors [10]. Such tactics are found prevalently across many software systems [20].

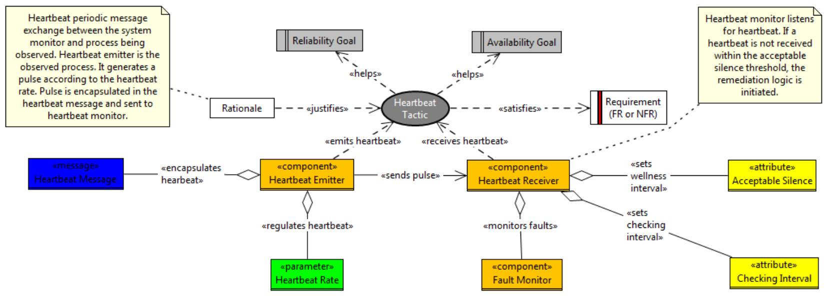
3.1 Tactical Spikes

Archie is built around the fundamental concept of *Tactical Spikes* which advocates iterative incremental implementation of quality concerns through adoption of individual tactics. For each development cycle, developers focus on a limited number of quality attributes. Tactical Spikes break the development of architecture and satisfaction of quality concerns into small sprints with each focusing on a specific quality attribute and a fine grained design decision which can be implemented independently from other choices. Satisfying quality concerns through Tactical Spikes is not a new idea and has been widely used in incremental design where the architecture emerges through several development cycles.

Although experienced developers are often fully aware of tactics that can be used to implement quality concerns, inexperienced developers tend to ignore the importance of quality attributes or adopt solutions that do not match the context of a problem [19]. To integrate agile design thinking into a developer's daily activities, Archie utilizes a knowledge base of architectural tactics and uses this resource to provide suggestions about architectural tactics which can be implemented to address each quality concern. Developers can search the catalogue of quality attributes in Archie and find associated architectural tactics which can be used to implement them. This provides an embedded design resource in the programmer's IDE.

3.2 Tool Support

To implement the idea of Tactical Spikes, we first introduce the fundamental concept of a Tactic Traceability Pattern (TTP) [16, 18] and then discuss the related functionality. TTP was previously used in maintaining the architecture of safety critical systems. It captures the context that a tactic can be used in, the design knowledge about the tactic, and primary conceptual roles to implement an architectural tactic. For the *heartbeat* tactic (Figure 2), the TTP indicates that this tactic can be used to address reliability and availability goals. The primary roles of this tactic include *emitter*, *receiver*, and *health monitor*.


Heartbeat periodic message exchange between the system monitor and process being observed. Heartbeat emitter is the observed process. It generates a pulse according to the heartbeat rate. Pulse is encapsulated in the heartbeat message and sent to heartbeat monitor.

Heartbeat monitor listens for heartbeat. If a heartbeat is not received within the acceptable silence threshold, the remediation logic is initiated.

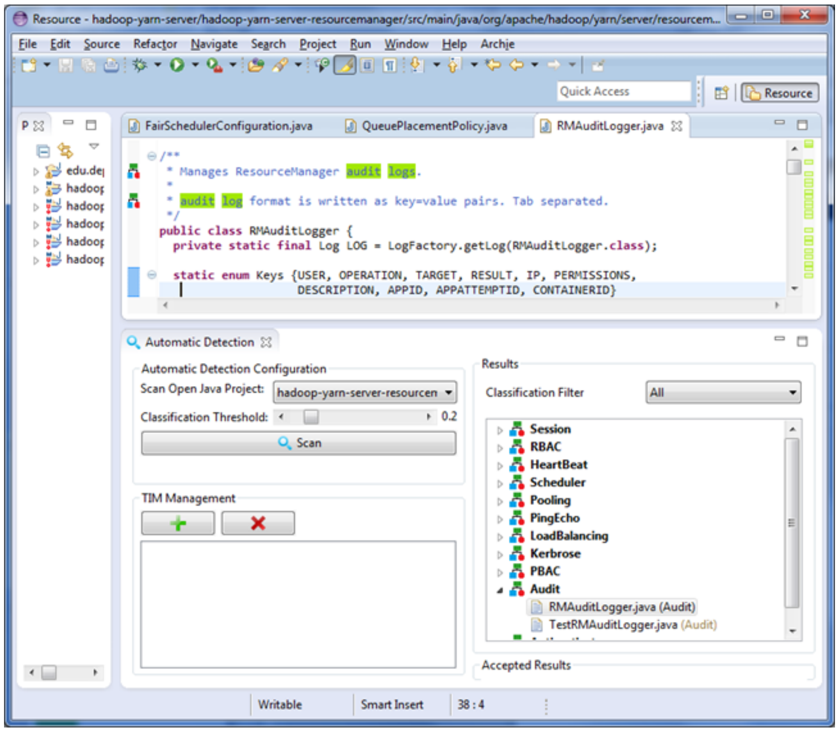
Fig. 2: HeartBeat Traceability Pattern: Helping Developer to Better Adopt and Implement a Tactic

Furthermore, 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. Lastly, a TTP also captures the underlying rationales for using the tactic which typically come in the form of a description that explains the quality concern being addressed. Each of the provided TTPs are 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.

The idea of Tactical Spikes are supported by TTP artifacts. Whenever developers want to implement a quality concern, they can look at the catalogue of TTPs in Archie and find the appropriate tactics which can be used to address their interested quality requirements. Furthermore, they would get an initial idea about which conceptual roles are involved when implementing the tactic. TTPs are modifiable artifacts (models), the developers can revise a TTP and document their micro design using basic UML modeling notations, they can also connect the tactical roles in the TTP to the source code. This can be done simply by selecting a segment of code and using mouse clicks to map it to either the entire TTP or to a specific role. Once the TTP is connected to the source code, it can provide a visual framework for communicating underlying architectural knowledge with other developers.

4 On Demand Detection of Tactical Spikes

Archie utilizes information retrieval and machine learning techniques to automatically detect architectural decisions from source code. The detection engine implements a set of classification techniques and each classifier is trained using code snippets representing different architectural tactics collected from hundreds of high-performance, open-source projects [17, 19, 20]. In the training phase the classifiers learn the terms, method and variable names, and development APIs that developers typically use to implement each tactic. These terms are used during the classification phase to calculate the likelihood that any given source file implements a tactic. The accuracy of the tactical spike detection technique is evaluated through several experiments which indicate our technique is capable of returning reliable results. The details of our detection engine, the classification formula and accuracy metrics are described in several previous works [17, 20].


Resource - hadoop-yarn-server/hadoop-yarn-server-resourcemanager/src/main/java/org/apache/hadoop/yarn/server/resourcem...
File Edit Source Refactor Navigate Search Project Run Window Help Archie
Quick Access
Resource
FairSchedulerConfiguration.java
QueuePlacementPolicy.java
RMAuditLogger.java
Automatic Detection
Automatic Detection Configuration
TIM Management
Results
Accepted Results
Scan
Writable
Smart Insert
38 : 4

```
/**
 * Manages ResourceManager audit logs.
 *
 * audit log format is written as key=value pairs. Tab separated.
 */
public class RMAuditLogger {
  private static final Log LOG = LogFactory.getLog(RMAuditLogger.class);

  static enum Keys {USER, OPERATION, TARGET, RESULT, IP, PERMISSIONS,
                    DESCRIPTION, APPID, APPATTEMPTID, CONTAINERID}
```

Scan Open Java Project: hadoop-yarn-server-resourcen
Classification Threshold: 0.2
Classification Filter: All

- Session
- RBAC
- HeartBeat
- Scheduler
- Pooling
- PingEcho
- LoadBalancing
- Kerbrose
- PBAC
- Audit
  - RMAuditLogger.java (Audit)
  - TestRMAuditLogger.java (Audit)

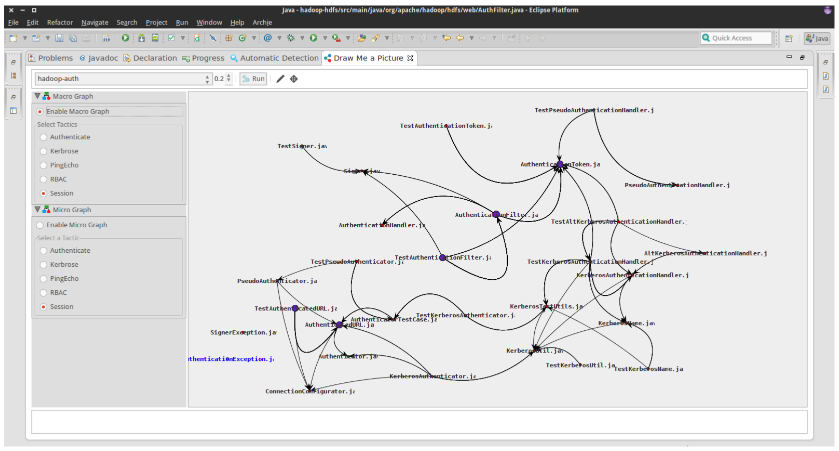
Fig. 3: Detected Tactical Spikes in Apache Hadoop Yarn Project

Figure 3 shows a screenshot of Archie where the classifiers are launched against the code in an Eclipse project. Several different architectural tactics are discovered in the source code of the Apache Hadoop Yarn project and categorized based on tactic type. Each category includes all the source files contributing to the implementation of each tactic. Furthermore, Archie highlights the method names, APIs and comments related to the tactic. For example, in Figure 3 comments about audit trails are highlighted which helps developers understand why this file is considered tactical.

This feature will help developers to better understand the current state of implemented tactical spikes. This is especially useful when a new developer joins the team and they need to quickly comprehend the architectural choices behind the source code or when developers alter a new module of software which they have not worked on previously. Furthermore, this auto-detection engine can assist developers in more easily documenting their previous tactical spikes. Initially they can detect and recover architectural tactics in the source code, and then use the previously described feature of TTPs and connect these source files into a visual and informative representation of tactics. The visualization and documentation of tactical spikes as TTPs increases program comprehension and design communication which significantly reduces the effort to maintain such models compared to traditional approaches.

5 Visualize Architecturally Significant Code

Architectural decisions often exhibit complex and cross-cutting impacts upon the design, therefore communicating this information textually or socially can be quite challenging. Archie's visualization feature helps developers understand underlying design decisions within the context of the design and/or code. This can be especially helpful in agile projects where *no time for training* is a common occurrence [11] and design knowledge is primarily shared through social mechanisms such as pair-programming and regular meetings. When using Archie, a developer can automatically generate a view of the system illustrating tactics implemented in the source code and the files involved in the implementation.



hadoop-auth 0.2

[✓ Enable Macro Graph]

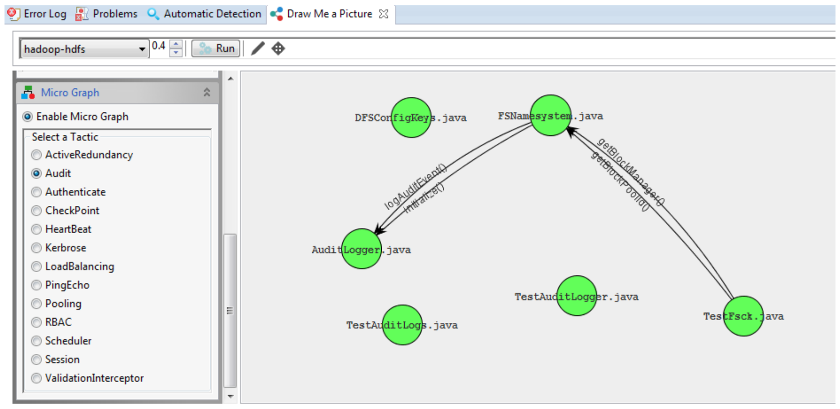
Select Tactics: [✓ Session]

Enable Micro Graph

Select a Tactic: [✓ Session]

Fig. 4: Visualization of Secure Session Management tactic Hadoop HDFS

Figure 4 depicts an automated visualization of secure session tactic in Hadoop HDFS subsystem. This view shows the source files involved in the implementation of tactic. In another example (Figure 5), we illustrate how Audit Trail tactic is implemented in Hadoop HDFS project, as well as the test files implemented to test this tactic.



hadoop-hdfs 0.4

[✓ Enable Micro Graph]

Select a Tactic
- ActiveRedundancy
- [✓ Audit]
- Authenticate
- CheckPoint
- HeartBeat
- Kerbrose
- LoadBalancing
- PingEcho
- Pooling
- RBAC
- Scheduler
- Session
- ValidationInterceptor

Fig. 5: Visualization of Audit Trail tactic and related test cases in Hadoop HDFS

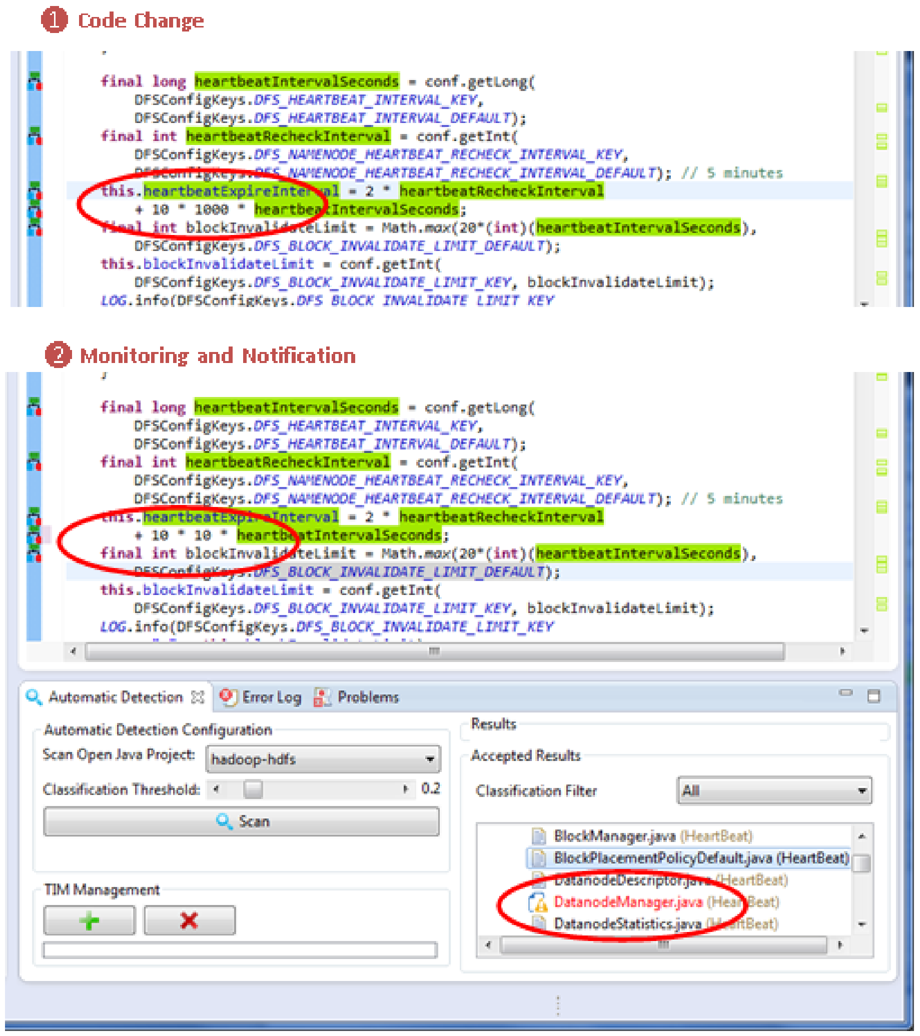
Archie provides features to help developers quickly create actionable models of tactical spikes. In this scenario, a developer may use Archie's automatic detection engine to locate implemented tactical spikes, and then connect these code snippets into TTPs. Establishing connections between source files and elements of TTPs requires only a single click. These features provide an easy and fast approach for creating rich architectural models during development. Furthermore, since these models are integrated with source code, any new programmer can open them in the programming IDE and understand the design decisions behind the code, the driving requirements, and the rationales behind design choices. Since TTPs can be customized, developers can add their own notes and elements into the model and communicate it with other developers on the team.

6 Decision Ownership and Accountability

Large software systems are typically created by numerous developers. Files may be created and maintained by a single developer, or through a collaborative effort by many. Code ownership may be defined as the individual(s) who are primarily responsible for the creation of an application component [25].

A single developer acting as the primary owner of a component has several potential benefits. The first is that code ownership leads to high quality results through pride of workmanship and is easier to hold individuals accountable for the creation and maintenance of the component. Research has also shown that more developers working on a single component typically leads to more defects [26]. However, owned code may suffer due to lack of external review since there may be less people examining it and may therefore hinder the discovery of problems [25].

In practice there are several different models of code ownership. As defined by fowler [8] the three levels of code ownership range from strong include, strong



```
final long heartbeatIntervalSeconds = conf.getLong(
    DFSConfigKeys.DFS_HEARTBEAT_INTERVAL_KEY,
    DFSConfigKeys.DFS_HEARTBEAT_INTERVAL_DEFAULT);
final int heartbeatRecheckInterval = conf.getInt(
    DFSConfigKeys.DFS_NAMENODE_HEARTBEAT_RECHECK_INTERVAL_KEY,
    DFSConfigKeys.DFS_NAMENODE_HEARTBEAT_RECHECK_INTERVAL_DEFAULT); // 5 minutes
this.heartbeatExpireInterval = 2 * heartbeatRecheckInterval
    + 10 * 1000 * heartbeatIntervalSeconds;
final int blockInvalidateLimit = Math.max(20*(int)(heartbeatIntervalSeconds),
    DFSConfigKeys.DFS_BLOCK_INVALIDATE_LIMIT_DEFAULT);
this.blockInvalidateLimit = conf.getInt(
    DFSConfigKeys.DFS_BLOCK_INVALIDATE_LIMIT_KEY, blockInvalidateLimit);
LOG.info(DFSConfigKeys.DFS_BLOCK_INVALIDATE_LIMIT_KEY
```

```
final long heartbeatIntervalSeconds = conf.getLong(
    DFSConfigKeys.DFS_HEARTBEAT_INTERVAL_KEY,
    DFSConfigKeys.DFS_HEARTBEAT_INTERVAL_DEFAULT);
final int heartbeatRecheckInterval = conf.getInt(
    DFSConfigKeys.DFS_NAMENODE_HEARTBEAT_RECHECK_INTERVAL_KEY,
    DFSConfigKeys.DFS_NAMENODE_HEARTBEAT_RECHECK_INTERVAL_DEFAULT); // 5 minutes
this.heartbeatExpireInterval = 2 * heartbeatRecheckInterval
    + 10 * 10 * heartbeatIntervalSeconds;
final int blockInvalidateLimit = Math.max(20*(int)(heartbeatIntervalSeconds),
    DFSConfigKeys.DFS_BLOCK_INVALIDATE_LIMIT_DEFAULT);
this.blockInvalidateLimit = conf.getInt(
    DFSConfigKeys.DFS_BLOCK_INVALIDATE_LIMIT_KEY, blockInvalidateLimit);
LOG.info(DFSConfigKeys.DFS_BLOCK_INVALIDATE_LIMIT_KEY
```

Scan Open Java Project: hadoop-hdfs
Classification Threshold: 0.2
Accepted Results
Classification Filter: All

```
BlockManager.java (HeartBeat)
BlockPlacementPolicyDefault.java (HeartBeat)
DatanodeDescriptor.java (HeartBeat)
DatanodeManager.java (HeartBeat)
DatanodeStatistics.java (HeartBeat)
```

Fig. 6: An architecture protection scenario

weak, and collective. Archie extends this notion to Design Ownership and provides an infrastructure for the developers to sign a tactical spike and be accountable for developing, testing, extending and maintaining related code snippets.

All architecturally significant code which has been mapped to TTPs or automatically detected by Archie are monitored for change activity. *Archie* integrates an event-based traceability engine where all the tactical classes are registered with the event server, and the owner of tactical spikes is registered as a subscriber. Whenever a user modifies architecturally significant code, a notification event is triggered and the owner is notified. The developer making changes is also sent a message in the warning list depicting a visual representation of the underlying architectural. Figure 6 depicts what the developer sees on 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. The heartbeat TTP is concurrently shown to the developer that code in *datanode.java* serves as the heartbeat emitter, and that it sends heartbeats to *HeartBeatManager*. Along with this refactoring a message is sent to the owner of Heartbeat tactic in the system. The owner is the developer who first implemented the tactic or identified themselves as the owner.

7 Related Work

Several other works have discussed the importance of design in agile software development. Wirfs-Brock [30] spoke about the significance of agile developers having the ability to quickly understand design problems and make timely decisions about how to address the issue.

While our work is innovative, there are a wide range of existing techniques that support agile software development. Spoelstra et al. [27] proposed a conceptual management tool for supporting software reuse in agile software development using a components derived from the Software Process Improvement (SPI) framework by Niazi et al. [22]. Their tool utilizes maturity levels and reuse practices in order to create a reuse maturity model. However, none were specifically focused on agile architecture development. There is a wide range of other tools supporting agile software development including tools for software project management and planning [7, 24], and modeling [5].

Several researchers have attempted to address architectural challenges through developing techniques for organizing, documenting or modeling architectural decisions. The Architecture Design Decision Support System (ADDSS) [6], Process based Architecture Knowledge Management Environment (PAKME) [2], and Architecture Rationale and Element Linkage (AREL) [29] are examples of these. However, these approaches are not adopted to the agile development mindset, and fail to address the scalability issues of managing potentially large numbers of architectural decisions. They also fail to connect design decisions to code, and/or provide little support for actually utilizing this knowledge during software maintenance.

8 Conclusion

While Archie's primary goal was to support secure software development, the automated features of Archie make it a unique tool for agile teams as well. The primary contribution of Archie is in the area of architectural centric support and preservation of architecture through detecting and tracing architectural concerns, and then using these trace links to keep developers fully informed of underlying architectural knowledge.

9 Acknowledgments

Archie was partially funded by the U.S. Department of Homeland Security.

References

1. Alves, E.L.G., Song, M., Kim, M.: Refdistiller: A refactoring aware code review tool for inspecting manual refactoring edits. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pp. 751–754. ACM, New York, NY, USA (2014)
2. Babar, M.A., Gorton, I.: A tool for managing software architecture knowledge. In: Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent, SHARK-ADI '07, pp. 11–. IEEE Computer Society, Washington, DC, USA (2007)
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 3rd edn. Addison-Wesley (2012)
4. Booch, G.: Draw me a picture. IEEE Software **28**, 6–7 (2011). DOI http://doi.ieeecomputersociety.org/10.1109/MS.2011.4
5. Buchmann, T.: Towards tool support for agile modeling: Sketching equals modeling. In: Proceedings of the 2012 Extreme Modeling Workshop, XM '12, pp. 9–14. ACM, New York, NY, USA (2012). DOI 10.1145/2467307.2467310
6. Capilla, R., Nava, F., Pérez, S., Dueñas, J.C.: A web-based tool for managing architectural design decisions. SIGSOFT Softw. Eng. Notes **31** (2006). DOI http://doi.acm.org/10.1145/1163514.1178644
7. Dhlamini, J., Nhamu, I., Kaihepa, A.: Intelligent risk management tools for software development. In: Proceedings of the 2009 Annual Conference of the Southern African Computer Lecturers' Association, SACLA '09, pp. 33–40. ACM, New York, NY, USA (2009). DOI 10.1145/1562741.1562745. URL `http://doi.acm.org.ezproxy.rit.edu/10.1145/1562741.1562745`
8. Folwer, M.: Codeownership. http://martinfowler.com/bliki/CodeOwnership.html (2006). URL `http://martinfowler.com/bliki/CodeOwnership.html`
9. van Gurp, J., Brinkkemper, S., Bosch, J.: Design preservation over subsequent releases of a software product: a case study of baan erp: Practice articles. J. Softw. Maint. Evol. **17**, 277–306 (2005). DOI 10.1002/smr.v17:4
10. Hanmer, R.: Patterns for Fault Tolerant Software. Wiley Series in Software Design Patterns (2007)
11. Justice, J.: For maximum awesome. http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=89889 (2014). URL `http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=89889`
12. Kruchten, P.: An ontology of architectural design decisions. Groningen Workshop on Software Variability management pp. 55–62 (2004)
13. Lee, L., Kruchten, P.: A tool to visualize architectural design decisions. In: QoSA, pp. 43–54 (2008)
14. Liang, P., Jansen, A., Avgeriou, P.: Knowledge architect: A tool suite for managing software architecture knowledge. In: Technical Report RUG-SEARCH-09-L01, University of Groningen (2009)
15. Martin, R.: Agile Software Development, Principles, Patterns, and Practices. Prentice Hall (2002)

16. Mirakhorli, M.: Tracing architecturally significant requirements: a decision-centric approach. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pp. 1126–1127. ACM, New York, NY, USA (2011). DOI 10.1145/1985793.1986014. URL `http://doi.acm.org/10.1145/1985793.1986014`
17. Mirakhorli, M.: Preserving the quality of architectural decisions in source code, PhD Dissertation, DePaul University Library (2014)
18. Mirakhorli, M., Cleland-Huang, J.: Using tactic traceability information models to reduce the risk of architectural degradation during system maintenance. In: Proceedings of the 2011 27th IEEE International Conference on Software Maintenance, ICSM '11, pp. 123–132. IEEE Computer Society, Washington, DC, USA (2011). DOI 10.1109/ICSM.2011.6080779
19. Mirakhorli, M., Mäder, P., Cleland-Huang, J.: Variability points and design pattern usage in architectural tactics. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pp. 52:1–52:11. ACM (2012)
20. Mirakhorli, M., Shin, Y., Cleland-Huang, J., Cinar, M.: A tactic centric approach for automating traceability of quality concerns. In: International Conference on Software Engineering, ICSE (1) (2012)
21. Moghadam, I.H., Ó Cinnéide, M.: Code-imp: A tool for automated search-based refactoring. In: Proceedings of the 4th Workshop on Refactoring Tools, WRT '11, pp. 41–44. ACM, New York, NY, USA (2011). DOI 10.1145/1984732.1984742
22. Niazi, M., Wilson, D., Zowghi, D.: A maturity model for the implementation of software process improvement: An empirical study. J. Syst. Softw. **74**(2) (2005)
23. Nongpong, K.: Integrating "code smells" detection with refactoring tool support. Ph.D. thesis, Milwaukee, WI, USA (2012). AAI3523928
24. Petersen, R.R., Wiil, U.K.: Asap: A planning tool for agile software development. In: Proceedings of the Nineteenth ACM Conference on Hypertext and Hypermedia, HT '08, pp. 27–32. ACM, New York, NY, USA (2008)
25. Rahman, F., Devanbu, P.: Ownership, experience and defects: A fine-grained study of authorship. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pp. 491–500. ACM, New York, NY, USA (2011)
26. Raymond, E.S.: The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly & Associates, Inc. (2001)
27. Spoelstra, W., Iacob, M., van Sinderen, M.: Software reuse in agile development organizations: A conceptual management tool. In: Proceedings of the 2011 ACM Symposium on Applied Computing, SAC '11, pp. 315–322. ACM (2011)
28. Tang, A., Avgeriou, P., Jansen, A., Capilla, R., Babar, M.A.: A comparative study of architecture knowledge management tools. Journal of Systems and Software **83**(3), 352 – 370 (2010). DOI DOI:10.1016/j.jss.2009.08.032
29. Tang, A., Jin, Y., Han, J.: A rationale-based architecture model for design traceability and reasoning. Journal of Systems and Software **80**(6), 918 – 934 (2007)
30. Wirfs-Brock, R.J.: Skills for the agile designer: Seeing, shaping and discussing design ideas. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, OOPSLA '10, pp. 323–326. ACM, New York, NY, USA (2010). DOI 10.1145/1869542.1869630