Architectural Clones: Toward Tactical Code Reuse

Mehdi Mirakhorli and Daniel E. Krutz
Software Engineering Department
Rochester Institute of Technology, Rochester, NY, USA
{mxmvse,dxkvse}@rit.edu

Abstract—**Architectural tactics are building blocks of software architecture. They describe solutions for addressing specific quality concerns, and are prevalent across many software systems. Once a decision is made to utilize a tactic, the developer must generate a concrete plan for implementing the tactic in the code. Unfortunately, this a non-trivial task for many inexperienced developers. Developers often use code search engines, crowdsourcing websites, or discussion forums to find sample code snippets. A robust tactic recommender system can replace this manual internet based search process and assist developers to reuse successful architectural knowledge, as well as implementation of tactics and patterns from a wide range of open source systems. In this paper we study several implementations of architectural choices in the open source community and identify the foundation of building a practical tactic recommender system. As a result of this study we introduce the concept of *tactical-clones* and use that as a basic element to develop our recommender system. While this NIER paper does not present the details of our recommender engine, instead it proposes the basis of our architecture recommender system.**

I. Introduction

The success of any complex software-intensive system is dependent on how it addresses quality attribute concerns of the stakeholders. Some include security, usability, availability, and interoperability. Designing a system to satisfy these concerns involves devising and comparing alternate solutions, understanding their trade-offs, and ultimately making a series of design choices. These architectural decisions typically begin with design primitives such as architectural tactics and patterns.

Tactics are the building blocks of architectural design [1], reflecting the fundamental choices that an architect makes to address a quality attribute concern. Because they are building blocks, tactics are composed together to form patterns. Architectural tactics come in many different shapes and sizes and describe solutions for a wide range of quality concerns. They are particularly prevalent across high-performance and/or fault tolerant software systems. Reliability tactics such as *redundancy with voting*, *heartbeat*, and *check pointing* provide solutions for fault mitigation, detection, and recovery; while performance tactics such as *resource pooling* and *scheduling* help optimize response time and latency.

The importance of implementing architectural tactics rigorously and robustly was highlighted by a small study we conducted as a precursor to this work. We investigated tactic implementations in Hadoop and OFBiz and evaluated their degree of stability during the maintenance process. For each of these projects we retrieved a list of bug fixes from the change logs (Nov. 2008 - Nov. 2011 for Hadoop, and Jan. 2009 - Nov. 2011 for OFBiz). Our analysis showed that tactics-related classes incurred 2.8 times as many bugs in Hadoop, and 2.0 times as many bugs in OFBiz as non-tactics related classes. In another preliminary work, we studied the implementation of security tactics in Chromium Browser and we observed that 10% of tactic implementations resulted in reported vulnerabilities. These observations suggest that tactic implementations, if not developed correctly, are likely to contribute towards the well-documented problem of architectural degradation [2]. Less experienced developers sometimes find this challenging, primarily because of the variability points that exist in a tactic, and the numerous design decisions that need to be made in order to implement a tactic in a robust and effective way. We found many examples of such questions on coding forums.

A robust tactic recommender system that shares sample code snippets from successful implementation of tactics in open source community can provide valuable support for the developers. Unfortunately, obtaining a practical tactic recommender system is not an simple task. This paper discusses the foundation of such a recommender system. Although there have been some initial development of source code recommender systems [3], [4], the primary focus of such works are on generic codes and not tactical codes. Therefore the challenges of obtaining and recommending architecturally significant code is still unexplored. Prior to proposing more specific guidelines for developing an architecture recommender system, we conducted an extensive study of architectural decisions in performance centric and dependable complex system. This study provided the foundation and motivation for the introduction of the concept of "Tactical Clones". We investigate this notion in several open source systems and use them as key elements in developing our recommender system.

II. Tactic's Implementation: Seen and Unseen

Our study involved review the source code of over 50 complex, performance centric open source systems including projects such as Google Chromium, Apache Hadoop, and Camel projects. For each of the studied projects we identified architecturally significant requirements, architectural tactics used to address them and source files used to implement tactics. As a result of this study we observed five issues that significantly influence development of any practical architecture recommender system. Each of these is discussed below:

• **No Single Solution.** There is no single way to address a quality requirements and also no single way to implement an architectural tactic. From one system to another system a tactic can be implemented entirely differently, this divergence is due to the differences in the context and constraints of each projects. For example, we reviewed the implementation of *heartbeat* tactic for reliability concerns in 25 different software systems. We observed the heartbeat tactic being implemented using (i) direct communication between the emitter and receiver roles found in *(Chat3 and Smartfrog systems)*, (ii) the observer pattern in which the receiver is registered as a listener to the emitter found in the *Amalgam system*, (iii) the decorator pattern 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 implementations which did not follow any documented design notion. Therefore a recommender system can not primarily rely on structural dependencies as a means of learning the best tactic implementation.

• **Structure Is Not a Key, But Impacts Quality.** Unlike design patterns, which tend to be described in terms of classes and their associations , tactics are described in terms of roles and interactions [1]. This means a tactic is not dependent upon a specific *structural* format. While a single tactic might be implemented using a variety of design notions or proprietary designs, the structural properties of tactical files can have significant on the quality of the tactic. Flaws such as cyclic dependencies, improper inheritance, unstable interfaces, and modularity violations are strongly correlated to increased bug rates and increased costs of maintaining the software. For example we found several security bug reports on Stack Overflow website where developers misuses *inheritance* relationship in implementation of *sandbox* tactic. In those cases a process outside sandbox had inheritance relationship with a process inside sandbox resulting in a breach into secure zones of the project. A recommender solution should take into account the internal quality of recommended code to avoid suggesting codes with design and structural flaws.

• **Tactical Clones: Right Level of Granularity.** While the implementation of tactics are different from one system to another system, the intrinsic characteristics of tactics are maintained across different projects. We call these as *architectural or tactical clones*. Based on our observation, tactical clones are the right level of granularity for recommending tactic implementations. In our code review process, we found that even for a simple tactic like heartbeat the implementation would result in a large number of interrelated files, each playing different roles such as heartbeat emitter, heartbeat receiver, configuration files to set heartbeat intervals and other parameters, supporting classes and interfaces to implement each tactical roles. More complex tactics, specially the cross-cutting ones can easily impact hundreds of source files. Therefore recommending code snippets for those tactics would create a large search space for the developers with lesser degree of reusability. The lack of structure, and a concrete micro-level design which can be recovered across multiple projects indicates that method level clones are the right level of granularity. In the next section of this paper we provide examples of such tactical clones.

• **Tactics Are Misused, Degraded or Implemented Incorrectly.** Open source repositories contain several cases where architectural tactics have been adopted by the developers without fully understanding the driving forces and variability points [5] associated with each tactic and consequences of implementing the tactic. The Heartbleed issue is a good example of such misuse. Heartbeat functionality in OpenSSL is an optional feature, while many developers could have easily disabled it in configuration files they fully ignored that. Furthermore, the implementation of heartbeat functionality did not followed solid software engineering practices.

In our analysis of bug reports in tactical fragments of the Hadoop project, we found that if a tactical file had a bug, then 89% of these issues were due to issues such as unhandled exceptions, type mismatches, or missing values in a configuration file. 11% of reports where due to wrong implementation. These bugs involved misconceptions in the use of the tactic, so that the tactic failed to adequately accomplish its architectural task. These kinds of bugs caused the system to crash under certain circumstances. For example, in one case a replication decision with a complex synchronization mechanism was misunderstood for different types of replica failure. Another example was a scheduling tactic which resulted in deadlock problem. This investigation shows that systems are exposed to new risks during implementation of the tactical decisions. A good tactic recommendation needs to take into account tactical code qualities, the context in which the tactics are adopted and the historical bug fixes and refactoring activities on candidate clone for recommendation. Our recommender system uses a set of static analysis tool to rank recommended tactical code clones based on their quality.

• **Object Oriented Metrics Are Not Indicator of Tactical Code Quality.** Our initial analysis of Chidamber and Kemerer's OO metrics [6] and tactical code snippets in Apache Hadoop and OfBiz systems indicates that tactical code snippets tend to relatively a have higher code complexity compared to non-tactical code snippets. For example implementing thread pooling requires devising solutions for thread safe problem which will results in a more complex implementation. Therefore OO metrics such as *WMC (Weighted Methods per Class)* or *CBO (Coupling Between Object classes)* can not solely be a good indicator of an improved tactical code snippet. A good tactic recommender system must take into account novel code metrics to filter potentially complex code samples which are difficult to comprehend and modify.

III. Architectural Clones: A Step Toward Tactical Code Reuse

In order to illustrate the concepts of architectural or tactical clones we conducted an explorative study and established a representative sample of such design clones. To do so we developed a semi-automated process for retrieving candidate instances of tactic-related classes then detected code clones across these tactical files. The process involved the following steps (1) building a software repository, (2) extracting instances of architectural tactics, (3) extracting code clones across projects, and finally (4) manually inspecting the results to investigate our hypothesis that tactical clones are a practical granularity for architectural reuse.

A. Building a software repository

We preselected 37 open source projects with a high number of architectural tactics. These tactic rich projects were identified through a previous study [7].

B. Extracting architectural tactics

To identify architectural tactics, we utilized a previously developed tactic detection algorithm and tool [7], [8]. This Tactic Detector's classifiers have been trained to detect architectural tactics such as *audit trail*, *asynchronous method invocation*, *authentication*, *checkpointing and roll back*, *heartbeat*, *role-based access control (RBAC)*, *resource pooling*, *scheduling*, *ping echo*, *hash-based method authentication*, *kerberos* and *secure session management*. Due to space constraints we provide only an informal description of our tactic detection approach; however a more complete description of the approach, including its related formulas, is provided in other publications [8], [9]. The tactic detection technique uses a set of classification techniques. These classifiers are trained using code snippets representing different architectural tactics, collected from hundreds of high-performance, open-source projects [5], [8], [9]. During the training phase, the classifier learns the terms (method and variable names as well as development APIs) that developers typically use to implement each tactic and assigns each potential indicator term a weight with respect to each type of tactic. The weight estimates how strongly an indicator term signals 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. During the classification phase, the indicator terms are used to evaluate the likelihood that a given file implements an architectural tactic.

The accuracy of the Tactic Detector has been evaluated in several studies [7]–[9]. In a series of experiments it was able to correctly reject approximately 77-100% of non-tactical code classes (depending on tactic types); recall 100% of the tactics-related classes with precision of 65% to 100% for most tactics tactics. The recall for the authentication, audit trail and asynchronous method invocation was 70% .

While this approach does not return entirely precise results, it has the a tuning parameter which enables us to only include the tactical files with higher prediction confidence in our analysis, which this will significantly reduce the search space and assist with the task of retrieving candidate tactical clones.

C. Detecting Tactical Clones

In order to detect architectural clones we used code clone detection techniques to identify reused tactical methods across different projects. We define the four types of tactical code clones using the definitions from Roy et al. [10] for code clones. Type-1 tactical clones are the simplest, representing identical tactical code except for variations in whitespace, comments, and layout to the type-4 clones, which are the most complex. Type-4 tactical clones, are tactical code segments that perform the same computation, but have been implemented using different syntactic variants. Our initial investigation indicates that type-4 or semantically equivalent tactical clones can be detected using complex code similarity techniques such as concolic and symbolic analysis [11]

Concolic analysis combines concrete and symbolic values to traverse all possible paths (up to a given length) of an application. Since concolic analysis is not affected by syntax or comments, identically traversed paths are indications of duplicate functionality, and therefore functionally equivalent code. These traversed paths are expressed in the form of *concolic output* which represents the execution path tree and displays the utilized path conditions and representative input variables. In order to detect tactical-clones we used a concolic analysis based clone detection technique [11], [12] on two type-4 clone examples examples of heartbeat are shown in Table I.

We then ran concolic analysis on these two code segments which produced the matching concolic output shown in Table II which indicated that original code snippets are tactical clone type-4. In this example, variable type integers are represented by a generic tag "SYMINT." Though not present in this example, other variable types are represented in a similar fashion in concolic output. Actual variable names do not appear anywhere in the output and are irrelevant to this clone detection process. We anticipate that open source repositories have a large number of tactical clone type-4 which can be used as items for a recommender system.

In an extensive experiment we ran a leading clone detection tool Nicad [10], over the tactical code snippets from 37 projects. We chose Nicad for our larger analysis since it is a more mature and refined tool than our experimental technique based upon concolic analysis. However, we believe that concolic analysis represents a more promising technique for tactical clone detection in our future work.

Table III shows tactics used in our study, as well as the number of tactical clones across projects (Note: We do not report tactical clones within the same project). Last column of this table illustrates total number of tactical files used in this study. The tactical clones were detected at method level, although we could have detected them as sub-method level we realized that method level tactical clones are easier to comprehend and therefore easier to reuse for the developers.

As a result of our explorative study we found several examples of identical tactical code snippets (type1,2 and 3) and several examples of conceptually equivalent tactical code snippets (type 4). Figure 1 shows the source code for RBAC tactic across two different projects. In this example two developers in different system have potentially developed the same code snippets to implement the tactic. This observation and several similar detected clones also emphasizes the fact that tactical clones are a more common granularity for code adoption and reuse.

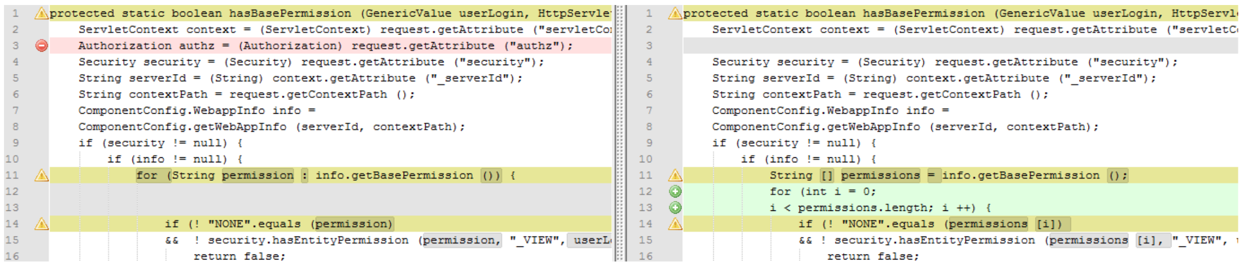
TABLE I
AN EXAMPLE HEARTBEAT TACTICAL CLONE

HeartBeat Example #1

```
boolean shouldBeRunning=true;
int smallInterval=10;
long lastHeartbeat=0;
int heartbeatInterval=10;
while (shouldBeRunning){
  Thread.sleep(smallInterval);
  if(System.currentTimeMillis()−lastHeartbeat>
    heartbeatInterval){
    sendHeartbeat();
    lastHeartbeat= System.currentTimeMillis();
  }
}
```

HeartBeat Example #2

```
long lastRunTime=0;
long timeSpan=System.currentTimeMillis();
long timeSinceLastRun=
  System.currentTimeMillis()−lastRunTime;
  if(timeSinceLastRun>10) {
    sendHeartbeat();
  lastRunTime = System.currentTimeMillis();
}
```



```
protected static boolean hasBasePermission (GenericValue userLogin, HttpServle
    ServletContext context = (ServletContext) request.getAttribute ("servletCo
    Authorization authz = (Authorization) request.getAttribute ("authz");
    Security security = (Security) request.getAttribute ("security");
    String serverId = (String) context.getAttribute ("_serverId");
    String contextPath = request.getContextPath ();
    ComponentConfig.WebappInfo info =
    ComponentConfig.getWebAppInfo (serverId, contextPath);
    if (security != null) {
        if (info != null) {
            for (String permission : info.getBasePermission ()) {


                if (! "NONE".equals (permission)
                && ! security.hasEntityPermission (permission, "_VIEW", userL
                    return false;
```

```
protected static boolean hasBasePermission (GenericValue userLogin, HttpServl
    ServletContext context = (ServletContext) request.getAttribute ("servletC

    Security security = (Security) request.getAttribute ("security");
    String serverId = (String) context.getAttribute ("_serverId");
    String contextPath = request.getContextPath ();
    ComponentConfig.WebappInfo info =
    ComponentConfig.getWebAppInfo (serverId, contextPath);
    if (security != null) {
        if (info != null) {
            String [] permissions = info.getBasePermission ();
            for (int i = 0;
            i < permissions.length; i ++) {
                if (! "NONE".equals (permissions [i])
                && ! security.hasEntityPermission (permissions [i], "_VIEW",
                    return false;
```

Fig. 1. Tactical Clones Detected in Two different Projects

TABLE II
DIFF OF HEARTBEAT CONCOLIC OUTPUT

Concolic Segment #1

```
### PCs: 1 1 0
a_1_SYMINT,
a_1_SYMINT,
    d1_2_SYMREAL,
a_1_SYMINT,
    d1_2_SYMREAL,
    s1_3_SYMSTRING,
```

Concolic Segment #2

```
### PCs: 1 1 0
a_1_SYMINT,
a_1_SYMINT,
    d1_2_SYMREAL,
a_1_SYMINT,
    d1_2_SYMREAL,
    s1_3_SYMSTRING,
```

TABLE III
DISCOVERED TACTICAL CLONES ACROSS 37 PROJECTS.

Tactic	Number of Clones	In Total Tactical Files
Audit	50	352
Authenticate	151	252
Checkpointing	8	138
Ping Echo	10	103
Pooling	1021	1073
RBAC	436	477
Scheduling	76	117
Secure Session	249	299
HeartBeat	0	11
Kerbrose	0	21

IV. CONCLUSION

In this preliminary work we investigated the challenges toward a robust and practical architecture recommender system. The notion of architectural clones can provide a reusable level of granularity for a recommender system. Our future research will extract tactical clones from thousands of open source system to build an architectural tactic recommender system.

ACKNOWLEDGEMENT

Partially sponsored by US Department of Homeland Security.

REFERENCES

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley, 2012.

[2] 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.

[3] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher, "Recommending source code for use in rapid software prototypes," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 848–858. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2012.6227134

[4] Y. Malheiros, A. Moraes, C. Trindade, and S. Meira, "A source code recommender system to support newcomers," in *Computer Software and Applications Conference, 2012 IEEE 36th Annual*, July 2012, pp. 19–24.

[5] M. Mirakhorli, P. Mäder, and J. Cleland-Huang, "Variability points and design pattern usage in architectural tactics," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. ACM, 2012, pp. 52:1–52:11.

[6] M. Hitz and B. Montazeri, "Chidamber and kemerer's metrics suite: a measurement theory perspective," *Software Engineering, IEEE Transactions on*, vol. 22, no. 4, pp. 267–271, Apr 1996.

[7] M. Mirakhorli, A. Fakhry, A. Grechko, M. Wieloch, and J. Cleland-Huang, "Archie: A tool for detecting, monitoring, and preserving architecturally significant code," in *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, 2014.

[8] 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.

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

[10] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, 2008, pp. 172–181.

[11] D. E. Krutz and E. Shihab, "Cccd: Concolic code clone detection," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, 2013.

[12] D. E. Krutz, "Concolic code clone detection," Ph.D. dissertation, Nova Southeastern University, 2012.