Architectural Clones: Toward Tactical Code Reuse

XXXXXXXXXXXXXXX
XXXXXXXXX
XXXXXXXXX
XXXXXXXXX
XXXXXXXXX, XX, XXX
XXXXXX@XXXXX.XXX

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 is a non-trivial task for even experienced developers. Developers often merely use search engines, crowd-sourcing websites, or discussion forums to find sample code snippets. A robust Tactic Search Engine can replace this manual internet based search process and help developers to reuse successful architectural knowledge, and properly implement tactics and patterns from a wide range of open source systems. In this paper we study several implementations of architectural tactics in the open source community and identify the foundation of building a practical Tactic Search Engine. As a result of this study, we introduce the concept of *tactical-clones* which may be used as the basic element of a tactic search engine.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

Keywords

Tactical Code Clone, Software Architecture, Code Reuse

1. INTRODUCTION

The success of any complex software-intensive system is dependent on how effectively it addresses the stakeholder's quality attribute concerns such as 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. 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 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 rigorously and robustly implementing architectural tactics was highlighted by a small study that was conducted in a previous work that investigated tactic implementations in Hadoop and OFBiz and evaluated their degree of stability during the maintenance process [14]. 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. These observations suggest that tactic implementations, if not developed correctly, are likely to contribute towards the well-documented problem of architectural degradation [22]. 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 search engine that shares sample code snippets from successful implementation of tactics in open source community can provide valuable support for the developers.

The primary contributions of this paper are:

1. Reporting the *results of a qualitative code review study* conducted to identify challenges in implementing architectural tactics and *reusing tactical code*.
2. Identifying the foundations of a practical tactic search engine.
3. Introducing the notion of *tactical clones* and formulating the next steps in realizing tactic search engine.

Although there have been some initial development of source code recommender systems [11, 12], the primary focus of these works are on generic code, and not tactical codes. Therefore the challenges of obtaining and recommending architecturally significant code is still unexplored. This paper focuses on identifying these challenges.

The structure of this paper is as the following. Section 2 present the underlying methodology used to conduct the qualitative study of tactic implementation. Section 3 discusses the results of our qualitative study, tactic implementation issues, reusability concerns and other observations across several implementation of architectural tactics. Furthermore this section summarizes the foundations for developing a practical tactic search engine. Section 4 presents

SAC'16, April 4-8, 2016, Pisa, Italy.

http://dx.doi.org/xx.xxxx/xxxxxxx.xxxxxxx ...$15.00.

the definitions of tactical clones and the process of extracting sample architectural clones from source code of several open source systems. Finally section 5 presents the future work and section 7 is the conclusion.

2. METHODOLOGY

Prior to proposing more specific guidelines for developing a tactic search engine, we conducted an extensive study of architectural decisions in performance centric and dependable complex system.

2.1 Goals

We believe that the current literature lacks insight into tactic's implementation. Revealing the low-level and non-abstract issues in coding architectural tactics can help shape the foundations of a tactic search engine.

2.2 Research Question

Our study was concerned about the following research high-level research question: RQ: What are the inherent characteristics of tactics implementations?

2.3 Project Selection

The following process was used to select a set of open source projects for this study.

- *Selection through Code Search:* The source code search engine *Koders* was used to search for the projects which have implemented a set of predefined architectural tactics. The search query for each tactic was composed from keywords used in the libraries that architect has previously used to implement the tactics. the results have been peer reviewed to ensure that each project has implemented the architectural tactic.
- *Selection by Meta-Data:* Project-related documents, such as design documents, online forums, etc. were searched for references and pointers to architectural tactics. This search was followed with a detailed source code search to ensure each identified project has the tactics implemented as well.

We have identified 40 open source projects using this process. The projects are elicited from different application domains, with diverse size, and developed using different programming languages. These dataset included projects such as Chromium, Apache Hadoop, Ofbiz and Hive which are comparable to industrial applications.

2.4 Study: Learning from the Trenches

For each of the studied projects we identified architecturally significant requirements, architectural tactics used to address them and source files used to implement tactics. For each of identified tactic, a peer-code review has been conducted to extract code snippets implementing the tactics. This was then followed by a manual reverse engineering process where our team members have utilized Enterprise Architect reverse engineering features to *draw a class diagram for each instance of the tactic*. In this study, two code reviewers with software architecture background were asked to document their observations of tactics implementations and formulates the challenges that impacts the development of a tactic search engine. The following section presents the results of this study.

3. QUALITATIVE STUDY

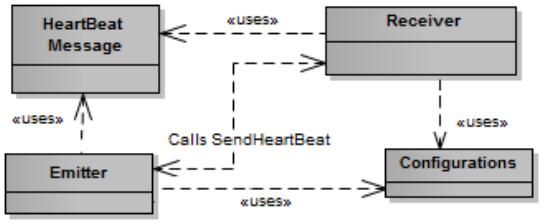
As a result of this study we observed five issues related to our research questions that can also significantly influence development of any practical tactic search engine:

3.1 No Single Solution

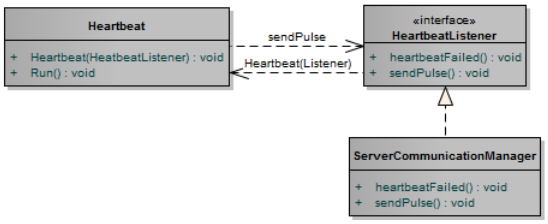
There is no single way to address a quality requirements and also no single way to implement an architectural tactic. From one system to another system a tactic can be implemented entirely differently, this divergence is due to the differences in the context and constraints of each projects.

For example, we reviewed the implementation of *heartbeat* tactic for reliability concerns in 20 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.

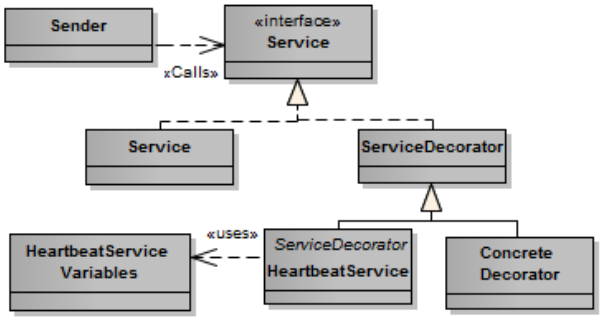
Therefore a tactic-search engine can not primarily rely on structural dependencies as a means of learning the best tactic implementation.



(a) HeartBeat with configuration files



(b) Observer design pattern to implement the tactic



(c) Decorator design pattern to implement the tactic

Figure 1: Hadoop : (a)Hadoop, Chat3, smartfrog (b)Amalgam System (c)Thera, JSRB, Rossume Systems

3.2 Structure Is Not a Key, But Impacts Code 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 imple-

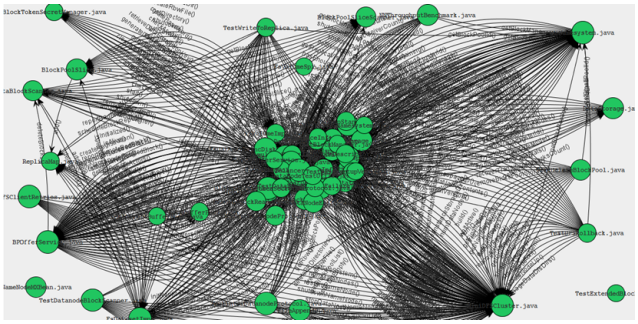


Figure 2: Resource Pooling Tactic Implemented in Apache Hadoop Project

mented 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.

Figure 2 visualizes the *resource pooling* tactic in Apache Hadoop project. Nodes in this graph represent the source file, and the edges are method calls between the source files. We observed several tactic implementations that did not exposed a well organized structure, typical they formed a full graph with several cyclic dependencies between each pair of files.

A tactic-search engine should take into account the internal quality of recommended code to avoid suggesting codes with design and structural flaws.

3.3 Tactical Clones, Right Level of Reuse-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 minimum reusable tactical features. 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.

3.4 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 [16] 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 practical tactic search engine 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.

3.5 Object Oriented Metrics Are Not Indicator of Tactical Code Quality

We performed a detained investigation of bug fixes in two of the systems included in our study. 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 [14]. 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 search engine must take into account novel code metrics to filter potentially complex code samples which are difficult to comprehend and modify.

4. ARCHITECTURAL CLONES: A STEP TOWARD TACTICAL CODE REUSE

the qualitative study conducted in this paper, motivated utilization of tactical clones as the minimum practical reuse granularity. In order to illustrate the concepts of architectural or tactical clones, our qualitative study was followed by an exploratory study where we 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.

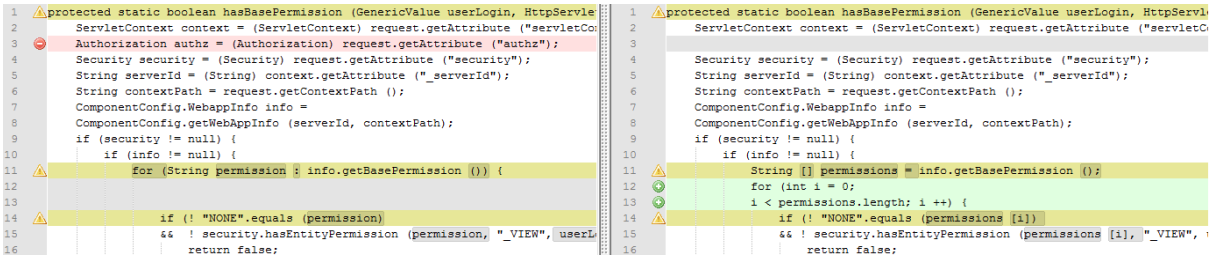
The process involved the following steps (1) building a software repository, (2) extracting instances of architectural tactics, (3) extracting code clones across projects, and finally (4) manually inspecting the results to investigate our hypothesis that tactical clones are a practical granularity for architectural reuse.

4.1 Building a software repository

We preselected 37 open source projects with a high number of architectural tactics.

4.2 Extracting architectural tactics

To identify architectural tactics, we utilized a previously developed tactic detection algorithm and tool [15, 17]. 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 con-*



```
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;
```

Figure 3: Tactical Clones Detected in Two different Projects

trol (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 [13,17]. 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 [13, 16, 17]. 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 [13, 15, 17]. 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.

4.3 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. [19] 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-2 tactical clones have variations in identifiers, types, whitespace, literals, layout, and comments, but are otherwise syntactically identical.

Type-3 tactical clones are tactical fragments which are copied and have modifications such as added or removed statements, variations in literals, identifiers, whitespace, layout and comments.

Type-4 tactical clones, are tactical code segments that perform the same computation, but have been implemented using different syntactic variants.

In an extensive experiment we ran a leading clone detection tool Nicad [19], 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 1: 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

4.4 Results

Table 1 shows tactics used in our study, as well as the number of tactical clones across projects. 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. We do not report tactical clones within the same project. Typically developers reuse the source code within a project. We were interested in the tactical clones reused across various projects so we could identify intrinsic and reusable tactic code snippets.

As a result of our exploratory study we found several examples of identical tactical code snippets. Most of the clones were type 1,2 and 3) but we also had several examples of *conceptually equivalent* tactical code snippets (type 4).

Figure 3 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.

5. FUTURE WORK

This paper provided insightful information about tactic implementation, code reuse and highlighted the path for building a practical tactic code search engine.

In the following subsections we present directions for future work.

5.1 Conceptually Equivalent Tactical Clones

Table 2: 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();
}
```

While tactical clone types 1, 2, and 3 primarily represent syntactically equivalent tactical code snippets reused across various projects, there are values in identifying, sharing and reusing tactical code snippets in form of clone type 4.

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 [10].

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 [8, 10] on two type-4 clone examples examples of heartbeat are shown in Table 2.

We then ran concolic analysis on these two code segments which produced the matching concolic output shown in Table 3 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 future work we plan to extend a primitive clone detection technique based on Concolic Analysis that is able to identify semantically equivalent code snippets. We will also augment this approach with text mining and information retrieval techniques.

5.2 Large Scale Study

Future work include a larger scale study where a few hundred open source projects will be studied to better understand how pervasive are tactical clones. Furthermore we will conduct a quantitative study to compliment the initial qualitative study reported in this paper. For each of the identified issue we will examine how frequent they happen across different implementation of tactics.

5.3 Developer Study

A series of experiments are required to rigorously evaluate the practical values of tactical clones in software reusability. In this regard a developer study can be conducted where developers can use a tactic-search engine to look for tactic implementations in terms of clones. Then the developers' feedback regarding the usefulness, reusability and practicality of retrieved tactical code is obtained.

Table 3: 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,
```

6. RELATED WORK

While no previous works have investigated architectural tactics as have, innumerable previous studies have analyzed code clones and their impact on software development. Juergens et al. [7] studied the consequences that code clones had on program correctness. This work found that commercial and open source software systems often suffer from inconsistent changes due to the presence of code clones, thus leading to possible system faults and increased maintenance costs. Many previous works have stated that code clones are undesirable since they often lead to more bugs and make their remediation process more difficult and expensive [4,18]. Other works have shown that clones may also substantially raise the maintenance costs associated with an application [7], the importance of which is highlighted by the fact that the maintenance phase of a project has been found to encompass between 40% and 90% of the total cost of a software project [20]. Ultimately, unintentionally making inconsistently applied bug fixes to cloned code across a software system increases the likeliness of further system faults [2].

Nicad is a powerful text-based hybrid clone detection technique, but there are numerous other popular clone detection tools and techniques. Some of which include Simian[1], CloneDR[2], MeCC[3], CCCD [9], and Simcad [21]. We are confident in our select of Nicad due to its effectiveness which has been demonstrated in previous research [19].

Although code clones have been demonstrated to be detrimental in certain situations, code reuse is imperative for most software development projects. Numerous previous works have studied software reuse on both open source, and commercial applications.

[1]http://www.redhillconsulting.com.au/products/simian/

[2]http://www.semdesigns.com/products/clone/

[3]http://ropas.snu.ac.kr/mecc/

Code reuse has been found to save significant time and resources for most projects, in addition to increasing the overall quality of the software [3]. Heinemann [5] performed an empirical study in 20 open source projects and analyzed 3.3 MLOC. Their analysis found that 9 of the 20 examined applications had software reuse rates of over 50%. Fortunately, most of the software reuse was through black-box reuse, and not through simply copying & pasting source code from the various applications. Mockus [**?**] conducted a study to determine the extent of software reuse in open source projects, identify the code that was being reused the most, and investigate patterns of large-scale reuse. This work found that 50% of the files were being used in more than one project, and that the most widely reused components were generally small, although some were comprised of hundreds of files.

Although there have been a few source code recommender systems [11, 12], the primary focus of these works are on generic source code, and not tactical code snippets. Therefore the challenges of obtaining and recommending architecturally significant code is still unexplored. This paper conducted a qualitative study and reported challenges related to implementation and reuse of tactical code snippets.

7. CONCLUSION

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

8. REFERENCES

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

[2] F. Deissenboeck, B. Hummel, and E. Juergens. Code clone detection in practice. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 499–500, New York, NY, USA, 2010. ACM.

[3] P. Devanbu, S. Karstu, W. Melo, and W. Thomas. Analytical and empirical evaluation of software reuse metrics. In *Software Engineering, 1996., Proceedings of the 18th International Conference on*, pages 189–199, Mar 1996.

[4] E. Duala-Ekoko and M. P. Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Trans. Softw. Eng. Methodol.*, 20(1):3:1–3:31, July 2010.

[5] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck. On the extent and nature of software reuse in open source java projects. In *Proceedings of the 12th International Conference on Top Productivity Through Software Reuse*, ICSR'11, pages 207–222, Berlin, Heidelberg, 2011. Springer-Verlag.

[6] M. Hitz and B. Montazeri. Chidamber and kemerer's metrics suite: a measurement theory perspective. *Software Engineering, IEEE Transactions on*, 22(4):267–271, Apr 1996.

[7] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, pages 485–495. IEEE Computer Society, 2009.

[8] D. E. Krutz. *Concolic Code Clone Detection*. PhD thesis, Nova Southeastern University, 2012.

[9] D. E. Krutz, S. A. Malachowsky, and E. Shihab. Examining the effectiveness of using concolic analysis to detect code clones. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1610–1615, New York, NY, USA, 2015. ACM.

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

[11] Y. Malheiros, A. Moraes, C. Trindade, and S. Meira. A source code recommender system to support newcomers. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, pages 19–24, July 2012.

[12] 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*, pages 848–858, 2012.

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

[14] M. Mirakhorli and J. Cleland-Huang. Modifications, tweaks, and bug fixes in architectural tactics. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 377–380, Piscataway, NJ, USA, 2015. IEEE Press.

[15] 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.

[16] 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*, FSE '12, pages 52:1–52:11. ACM, 2012.

[17] 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.

[18] M. Mondal, C. K. Roy, and K. A. Schneider. An empirical study on clone stability. *SIGAPP Appl. Comput. Rev.*, 12(3):20–36, Sept. 2012.

[19] 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*, pages 172–181, 2008.

[20] R. Shukla and A. K. Misra. Estimating software maintenance effort: a neural network approach. In *Proceedings of the 1st India software engineering conference*, ISEC '08, pages 107–112, New York, NY, USA, 2008. ACM.

[21] M. Uddin, C. Roy, and K. Schneider. Simcad: An extensible and faster clone detection tool for large scale software systems. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 236–238, May 2013.

[22] 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.