# Architectural Clones: Toward Tactical Code Reuse

Daniel E. Krutz and Mehdi Mirakhorl
Software Engineering Department
Rochester Institute of Technology
1 Lomb Memorial Drive
Rochester, NY, USA
{dkrutz, mehdi}@se.rit.edu

## ABSTRACT

Architectural tactics are the 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 resort to using 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 proper architectural knowledge and accurately implement tactics and patterns from a wide range of open source systems. In this paper we analyze several implementations of architectural tactics in the open source community and identify the foundation for building a practical Tactic Search Engine. We also introduce the concept of *tactical-clones* which may be used as the basic element of a tactic search engine.

## CCS Concepts

•**Software and its 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. Architectural tactics 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. 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 previous work [14] which 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). The previous analysis showed that tactic-related classes incurred 2.8 times as many bugs in Hadoop, and 2.0 times as many bugs in OFBiz as non-tactic related classes. These observations suggest that tactic implementations, if not correctly developed, are likely to contribute towards the well-documented problem of architectural degradation [24]. Less experienced developers sometimes find this challenging, primarily due to 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. A robust tactic search engine which can share sample code snippets from successful implementations of tactics in the open source community can be invaluable for developers.

Our primary contributions in this paper are:

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

Although there has been some initial development of source code recommender systems [11, 12], the primary focus of this previous research is only on retrieving generic functional code and not tactical code. Therefore the challenges of obtaining and recommending tactical code is still unexplored. This paper focuses on identifying these challenges.

This paper organized as follows: Section 2 presents the underlying methodology used to conduct the qualitative study of tactic implementations. Section 3 discusses the results of our qualitative study, tactic implementation issues, reusability concerns and other observations across several implementation of architectural tactics. This section also summarizes the foundations for developing a practical tactic search engine. Section 4 presents the definitions of tactical clones and the process of extracting sample architectural clones from source code of several open source systems. Section 5 presents possible future work to build on our research and Section 6 discusses related works. Section 7 concludes our paper.



*SAC 2016, April 04-08, 2016, Pisa, Italy*



DOI: http://dx.doi.org/10.1145/2851613.2851787

## 2. METHODOLOGY

Prior to proposing more specific guidelines for developing a tactic search engine, we conducted an extensive study of architectural decisions and their implementations in 40 performance centric, safety critical dependable systems [16].

### 2.1 Goals

Current research lacks insight into the implementation of tactics. Revealing low-level and non-abstract issues in developing architectural tactics can help shape the foundation of a tactic search engine.

### 2.2 Research Question

Our study is focused on the following research question: *What are the implementation characteristics of architectural tactics that are reflected in the source code?*

### 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 locate projects which have implemented a set of predefined architectural tactics. The search query for each tactic is composed of keywords used in the libraries that the 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 and online forums were searched for references and pointers to architectural tactics. This search was followed by a detailed source code review to ensure that each identified project has the tactics implemented.

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

### 2.4 Study: Learning from the Trenches

For each of the examined projects we identified architecturally significant requirements, architectural tactics used to address them, and source files used to implement tactics. For all identified tactics, a peer-code review was conducted to extract code snippets implementing the tactics. This was followed by a manual reverse engineering process where our team members utilized Sparx Enterprise Architect[1] reverse engineering features to draw a class diagram for each instance of the tactic. In this study, two code reviewers who are experienced in software architecture were asked to document their observations of tactic implementations and formulate the challenges that impacts the development of a tactic search engine.

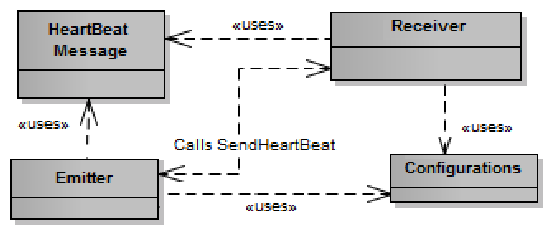
## 3. QUALITATIVE STUDY

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

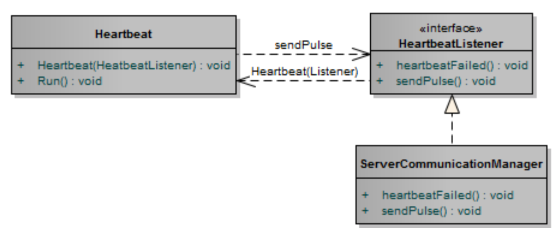
### 3.1 No Single Solution

There is no single way to address quality requirements and also no single way to implement an architectural tactic. System tactics may be implemented entirely differently from one system to another. This divergence is due to the differences in the context and constraints of each project. For example, we reviewed the implementation of the *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 where 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 designs.

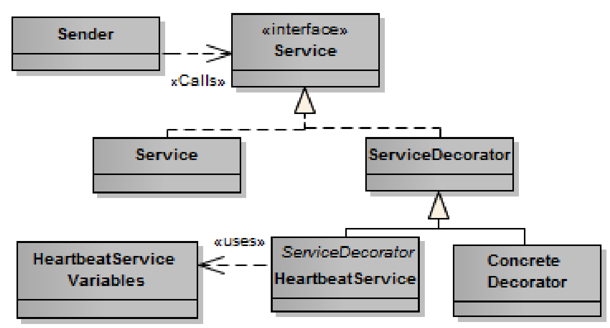
*Therefore a tactic-search engine can not primarily rely on structural dependencies as a means of learning the best tactic implementation.*




(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 that 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 a significant impact on the quality of the tactic. Flaws such as cyclic dependencies, improper inheritance, unstable interfaces, and modularity violations

[1] http://www.sparxsystems.com/products/ea/downloads.html

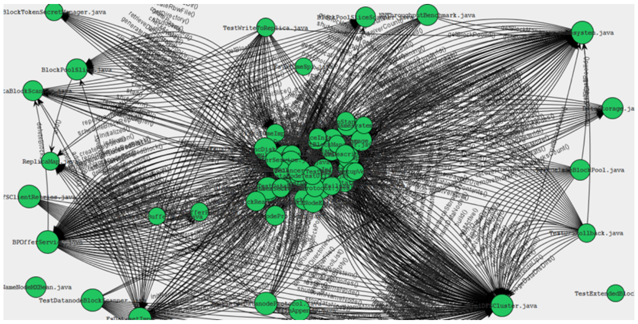


**Figure 2: Resource Pooling Tactic Implemented in Apache Hadoop Project**

are strongly correlated to increased bug rates and the elevated maintenance costs.

Figure 2 displays the *Resource Pooling* tactic in the Apache Hadoop project. Nodes in this graph represent the source file, and the edges are method calls between the source files. We observed several such tactic implementations that did not expose a well organized code structure. Typically, the source files in the tactic form 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 code with design and structural flaws.*

## 3.3 Tactical Clones, Right Level of Reuse-Granularity

While the implementation of tactics vary between different systems, the *intrinsic characteristics of tactics are maintained across different projects*. These are known as *architectural or tactical clones*. Based on our observations, 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, with each playing a different role. Some of which include *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 we provide examples of such tactical clones.

## 3.4 Tactics Are Misused, Degraded or Implemented Incorrectly.

Open source repositories contain numerous cases where architectural tactics have been adopted by developers without them fully understanding the driving forces and variability points associated with each tactic and consequences of implementing the tactic [16]. The Heartbleed issue is a good example of such a misuse. Heartbeat functionality in OpenSSL is an optional feature, although many developers have ignored this option.

In our analysis of bug reports in tactical fragments of the Hadoop project, we found that when a tactical file had a bug, then 89% of these concerns were due to issues such as unhandled exceptions, type mismatches, or missing values in a configuration file. Incorrect implementations led to 11% of all reports. These bugs involved misconceptions in the use of the tactic, so that the tactic failed to adequately accomplish its underlying architectural goals. These types 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 failures. Another example was a scheduling tactic which resulted in a 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, and the context in which the tactics are adopted, Additional quality factors include the history of bug fixes and refactoring activities on the candidate clones.*

## 3.5 Object Oriented Metrics Are Not Indicator of Tactical Code Quality

We performed a detailed investigation of bug fixes in two of the systems included in our study. Our initial analysis of the OO metrics of Chidamber et al. [6] and tactical code snippets in Apache Hadoop and OfBiz indicates that tactical code snippets typically have a relatively higher code complexity compared to non-tactical code snippets [14]. For example, implementing *thread pooling* requires devising solutions for the *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 novel code metrics into account to filter potentially complex code samples. Such metrics should help filter code snippets which are difficult to comprehend and modify.*

# 4. ARCHITECTURAL CLONES: A STEP TOWARD TACTICAL CODE REUSE

The qualitative study we conducted motivated the utilization of tactical clones as an appropriate level of granularity in respect to code reusability. 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 these design clones. To assist, we developed a semi-automated process for retrieving candidate instances of tactic-related classes then detected code clones across these tactical files.

The process of detecting tactical clones involves the following steps: (1) building a software repository, (2) extracting instances of architectural tactics, (3) extracting code clones across projects, and (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

To build our repository of software systems, we preselected 37 open source projects with a high number of architectural tactics.

## 4.2 Extracting architectural tactics

We utilized a previously developed tactic detection algorithm and tool [15, 17] to identify architectural tactics. These Tactic Detector's classifiers have been trained to find 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

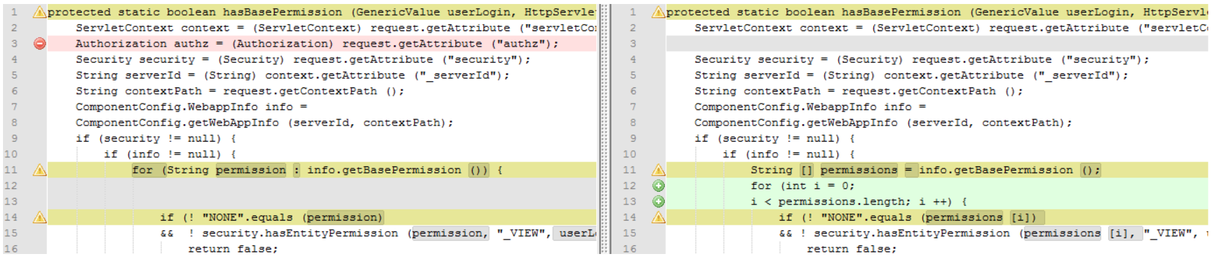



**Figure 3: Tactical Clones Detected in Two different Projects**

description of our tactic detection approach. However a more complete description of the approach, including its related formulas, may be found in previous 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 perfectly precise results, it has a tuning parameter which enables us to only include the tactical files with higher prediction confidence in our analysis, which 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 by extending the definitions from Roy et al. [21].

**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 [21], over the tactical code snippets from 37 projects. We chose Nicad for our analysis since it is a mature and refined tool which has demonstrated its effectiveness in previous research [20].

**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. The last column of this table illustrates the total number of tactical files used in our analysis. The tactical clones were detected at the method level. While we could have detected tactical clones at the 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 since developers typically 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 tactical code snippets. As a result of our exploratory study we found several examples of identical tactical code snippets. While most of the clones were type 1, 2 and 3, we also had several examples of *conceptually equivalent* tactical code snippets (type 4).

Figure 3 shows the source code for RBAC tactics across two different projects. In this example two developers in different systems have potentially created the same code snippets to implement the tactic. This example is one instance among several similar observations of tactical clones across different projects. This supports the hypothesis that tactical clones are an appropriate level of granularity in respect to code reusability. However, as stated in future work we plan to examine this hypothesis in set of developer studies.

# 5. FUTURE WORK

This paper provides crucial information about tactic implementation and code reuse. However, there is still future work to be conducted in this area.

## 5.1 Conceptually Equivalent Tactical Clones

While tactical clone types 1, 2, and 3 primarily represent syntactically equivalent tactical code snippets reused across various projects, sharing and reusing tactical code snippets that are type 4 clones would be very beneficial. Our initial investigation indicates that type-4 or semantically equivalent tactical clones can be detected using complex code similarity techniques such as symbolic

**Table 2: An Example HeartBeat Tactical Clone**

| HeartBeat Example #1 | HeartBeat Example #2 |
|---|---|
| `boolean shouldBeRunning=true;`<br>`int smallInterval=10;`<br>`long lastHeartbeat=0;`<br>`int heartbeatInterval=10;`<br>`while (shouldBeRunning){`<br>`  Thread.sleep(smallInterval);`<br>`  if(System.currentTimeMillis()-lastHeartbeat>`<br>`    heartbeatInterval){`<br>`    sendHeartbeat();`<br>`    lastHeartbeat= System.currentTimeMillis();`<br>`  }`<br>`}` | `long lastRunTime=0;`<br>`long timeSpan=System.currentTimeMillis();`<br>`long timeSinceLastRun=`<br>`  System.currentTimeMillis()-lastRunTime;`<br>`  if(timeSinceLastRun>10) {`<br>`    sendHeartbeat();`<br>`  lastRunTime = System.currentTimeMillis();`<br>`}` |

and concolic analysis [8, 10].

Concolic analysis combines concrete and symbolic values to traverse all possible paths 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 type-4 clones. 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. This can be very beneficial for the type-4 clone detection process. We anticipate that open source repositories have a large number of tactical type-4 clones which can be used as input for a tactic search engine.

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 should also include a larger scale study where at least several hundred open source projects will be studied to better understand how pervasive tactical clones are. We will also conduct a quantitative study to compliment our initial qualitative study reported in this paper. For each of the identified issues, we will examine how frequently they occur across different implementations of tactics.

## 5.3 Developer Study

A series of experiments are required to rigorously evaluate the practical value of tactical clones in software reusability. A study may be conducted where developers can use a tactic-search engine to look for tactic implementations in terms of clones. The developer feedback regarding the usefulness, reusability and practicality of retrieved tactical code would then be collected.

# 6. RELATED WORK

While no previous works have investigated architectural tactics as we have, numerous 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, 19]. Other research has 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 [22]. 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[2], CloneDR[3], MeCC[4], CCCD [9], and Simcad [23]. We are confident in our selection of Nicad due to its effectiveness which has been demonstrated in previous research [21].

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. Code reuse has been found to save significant time and resources for most projects, along with 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 reuse was through black-box reuse, and not through simply copying & pasting source code from the various applications. Mockus [18] conducted a study to deter-

[2]http://www.redhillconsulting.com.au/products/simian/
[3]http://www.semdesigns.com/products/clone/
[4]http://ropas.snu.ac.kr/mecc/

**Table 3: Diff of HeartBeat Concolic Output**

| Concolic Segment #1 | Concolic Segment #2 |
|---|---|
| `### PCs: 1 1 0`<br>`a_1_SYMINT,`<br>`a_1_SYMINT,`<br>`    d1_2_SYMREAL,`<br>`a_1_SYMINT,`<br>`    d1_2_SYMREAL,`<br>`    s1_3_SYMSTRING,` | `### PCs: 1 1 0`<br>`a_1_SYMINT,`<br>`a_1_SYMINT,`<br>`    d1_2_SYMREAL,`<br>`a_1_SYMINT,`<br>`    d1_2_SYMREAL,`<br>`    s1_3_SYMSTRING,` |

mine the extent of software reuse in open source projects, identify the most reused code, 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 work we investigated the challenges toward creating a robust and practical tactic search engine. The study revealed issues related to coding architectural tactics. Furthermore, our study suggests that the notion of architectural clones can provide a reusable level of granularity for a tactic search engine. Future work will examine this suggestion through a rigorous developer study.

# 8. ACKNOWLEDGMENTS

The work in this paper was partially funded by the US National Science Foundation grant #CCF-1543176.

# 9. 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.

[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] A. Mockus. Large-scale code reuse in open source software. In *Emerging Trends in FLOSS Research and Development, 2007. FLOSS'07. First International Workshop on*, pages 7–7. IEEE, 2007.

[19] 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.

[20] C. K. Roy and J. R. Cordy. An empirical study of function clones in open source software. In *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*, pages 81–90. IEEE, 2008.

[21] 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.

[22] 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.

[23] 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.

[24] 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.