Keecle: Mining Key Architecturally Relevant Classes using Dynamic Analysis

Liliane do Nascimento Vale[1,2] and Marcelo de A. Maia[1]
[1]Faculty of Computing - Federal University of Uberlândia, Uberlândia – MG – Brazil
[2]Computer Science Department - Federal University of Goiás, Catalão – GO – Brazil
Email: liliane.ufg@gmail.com, marcelo.maia@ufu.br

***Abstract*—Reconstructing architectural components from existing software applications is an important task during the software maintenance cycle because either those elements do not exist or are outdated. Reverse engineering techniques are used to reduce the effort demanded during the reconstruction. Unfortunately, there is no widely accepted technique to retrieve software components from source code. Moreover, in several architectural descriptions of systems, a set of architecturally relevant classes are used to represent the set of architectural components. Based on this fact, we propose Keecle, a novel dynamic analysis approach for the detection of such classes from execution traces in a semi-automatic manner. Several mechanisms are applied to reduce the size of traces, and finally the reduced set of key classes is identified using Naïve Bayes classification. We evaluated the approach with two open source systems, in order to assess if the encountered classes map to the actual architectural classes defined in the documentation of those respective systems. The results were analyzed in terms of precision and recall, and suggest that the proposed approach is effective for revealing key classes that conceptualize architectural components, outperforming a state-of-the-art approach.**

I. Introduction

The cost and effort needed to understand and adapt internal elements of software systems is related to the investigation of artifacts such as source code and documentation. Moreover, in many cases, documentation concerning design decisions is missing, or when it exists, it is neither updated nor complete. In that case, developers are required to analyze the source code, which is the only source of reliable information to understand the software architecture. Traditionally, software architectures are documented in a package based structure, since it is easier to map to actual artifacts. However, quite often this is not the best architectural organization [1], and when the architectural documentation is available, it is often outdated because of phenonema, such as, architectural drift or erosion [2]. To alleviate these problems, several architecture reconstruction techniques have been proposed [3], but a number of problems hinder the use of these techniques.

A recent study performed a comparative analysis to measure the accuracy of six different architectures recovery techniques using eight ground-truth architectures, which indicated that the limitations of these techniques are related to the accuracy, to the conditions under which the techniques succeed or fail, to the number and size of selected systems etc. For instance, the average accuracy using the MoJoFM measure [4] was 45%.

In a work that closely relates to ours, Zaidman and Demeyer [5] proposed a technique which can identify the most important classes in a system—the key classes. They characterized these key classes as typically having a lot of "control" within the application. In order to find these "controller classes", they presented a detection approach that is based on dynamic coupling and webmining, obtaining a precision of around 50%. In our approach, the concept of "key classes" can be mapped to "architecturally relevant classes", i.e., those that are central for defining the meaning of an architectural component.

Our goal is to provide an approach with higher accuracy that recovers execution trace subtrees whose roots are calls to methods for those key classes. Our hypothesis is that architectural components can be matched to subtrees of execution traces that have a larger number of distinct method calls, which are typically near to the main tree root, i.e., they are low-depth nodes in the method call tree. Several tools that implement this approach are used to: capture traces; compress traces; discard identical subtrees; and find the architectural relevant classes using Naïve Bayes classification algorithm.

The contributions of this paper are twofold: 1) we propose Keecle, a novel technique for the identification of architecturally relevant classes of a software system that can be provided for developers as a high-level overview to help understanding and maintenance activities; 2) we provide a preliminary empirical evaluation of Keecle using two open source systems used in [5] showing that it outperforms previous work.

This paper is organized as follows. Section II shows an overview of the Keecle approach, describing how execution traces are captured, compressed, and transformed into more compact subtrees, and also how the key classes are mined from those subtrees. Section III presents the study setting to evaluate the accuracy of the Keecle approach and the evaluation results. Some related works are presented in Section IV. Finally, the last section presents the conclusion and future work.

II. The proposed approach

The ideal solution of interest would be the identification of architectures based on components from the source code. However, because it is difficult to identify those components in large systems, we alternatively propose, as a starting point, the semi-automatic identification of architecturally relevant classes (key classes) that represent components. Tahvildari and Kontogiannis [6] defined key classes as:

"... the classes that implement the key concepts of a system. Usually, these most important concepts of a system are implemented by very few key classes, which can be characterized by a number of properties. These classes which we called key classes manage a large amount of other classes or use them in order to implement their functionality."

Their idea that *very few key classes* implement the concepts of a system motivated us to match this notion of key classes as those that are typically used by developers to explain a software architecture. So, our goal is to semi-automatically detect these key classes from execution traces and evaluate their occurrence in architecture-oriented documentation of subject systems. In general these classes have strong control over the system and rely on other classes to implement software features. Our definition is that architecturally relevant classes – *key classes* – are represented by method calls, structured in call trees constructed during the system execution. The key classes are expected to be near the roots (or subroots) of large execution trace subtrees that contain a large number of method calls (nodes) from different classes and packages.

We will provide an overview of the proposed approach, aided by a set of tools, organized into three phases which are presented in the following subsections.

A. Phase 1 - Capturing Traces

We use the term *feature* as a functionality that can be described from the user's point of view or as an observable behavior of the system that can be triggered by the user [7]. Whenever developers aim at comprehending software internals, we expect that they already know its main features. Our approach, as any other based on dynamic analysis, requires choosing representative features from documentation that are expected to cover all components of interest. The empirical evaluation can verify if those selected features are able to recover the largest possible number of classes in relation to total number of system's classes.

The target system is instrumented with an AspectJ-based tool to collect the executed methods. During the execution scenario of each feature, trace files are created for each triggered thread. Each line of the trace file corresponds to a method call, which has the name of the qualified method, class and package, moreover there is the corresponding level in the call stack that enables the construction of a method call tree.

B. Phase 2 - Reducing the Size of the Traces

In this section, we present the three steps taken in the trace-reduction process. Algorithm 1 is the pseudocode for extracting reduced subtrees from execution traces.

1) ***Compressing Traces****:* Traces are compressed removing parts that are identical, typically due to loops or recursion in method calls [8]. So, the expected result of this compression is that the resulting larger subtrees contain more calls to distinct methods, instead of an absolute higher number of calls that could represent a high number of calls to a few distinct methods. Due to the fact that the proposed approach filters subtrees based on their size, as will be explained, means this compression phase has an important meaning.

Algorithm 1: Trace Reduction Process for a Set of Trace File *TF*

```
   Input: A set of trace files TF;
1  init
2      subTreeList ← TraceCompressor(TF)
3      halfDepth ← maxDepth(subTreeList) ÷ 2
4      NISubtreeExtractor(subTreeList,greatestSubTree(subTreeList),1,halfDepth)
5      IdenticalSubtreeFilter(subTreeList)
6      return subTreeList
7  function NISubtreeExtractor(subTreeList, greatestST, maxExpandedLevel,
   halfDepth)
8      if (maxExpandedLevel < halfDepth) then
9          subTreeList.remove(greatestST)
10         subTreeList.add(greatestST.children())
11         if (maxExpandedLevel < subTreeTarget.level + 1) then
12             maxExpandedLevel ← greatestST.level + 1
13         NISubtreeExtractor(subTreeList, greatestSubTree(subTreeList),
14                 maxExpandedLevel, halfDepth)
```

2) ***Extracting Non-intersecting Trace Subtrees****:* Our rationale is that the root node of a subtree or subroots near to the root are more likely to indicate a key class that represents an architectural component. Moreover, nodes near the leaves of the subtree are more likely to represent fine-grained, not architecturally relevant actions, although we agree that exceptions may occur.

The proposed method is based on the extraction of large subtrees without a non-proxy root. Proxy roots are those with only one child (method call) or important children (large subtree) and other non-important children (small subtrees). For each level, a set of subtrees' roots are analyzed based on their size and the size of their children.

The subtree recursive analysis process descends trace tree, identifying and extracting subtrees. The algorithm selects firstly the largest subtrees to try to split it in smaller subtrees. The recursive process ends when there is a subtree with a root in the level that corresponds to half of the original trace tree depth. When half the level corresponds to a non-integer value, the value is rounded to a higher value. This choice for stop-criterion was based on the observation that subtrees of interest, typically are large and its roots or subroots are near to the original tree root.

Figure 1 shows an example of this extracting process. It shows a *trace tree* where nodes are method calls. The trace tree has size 20 and contains six levels. In this case, the limit for the expansion process is at level 3, because of the defined threshold half the maximum height.

Starting from the root *Bootstrap.start*, three child subtrees are identified with sizes: *2, 10* and *7* respectively, from the left to right on level 2. The first subtree (green subtree) has root *LogFactory, getLog*, and we named it subtree *A*. The second subtree (red subtree) has root *Catalina.start*, and we named it subtree *B*. The third subtree (blue subtree) has root *LogFactory.getLog*, and we named it subtree *C*.

The roots of the new subtrees are on level 2, then the extraction process continues on level 3. The algorithm chooses the largest subtree to split it into smaller subtrees, namely the subtree *B* with size 10. When the subtree *B* is splitted, two new subtrees are analyzed (subtree *D* and *E*). The subtree *D* (orange

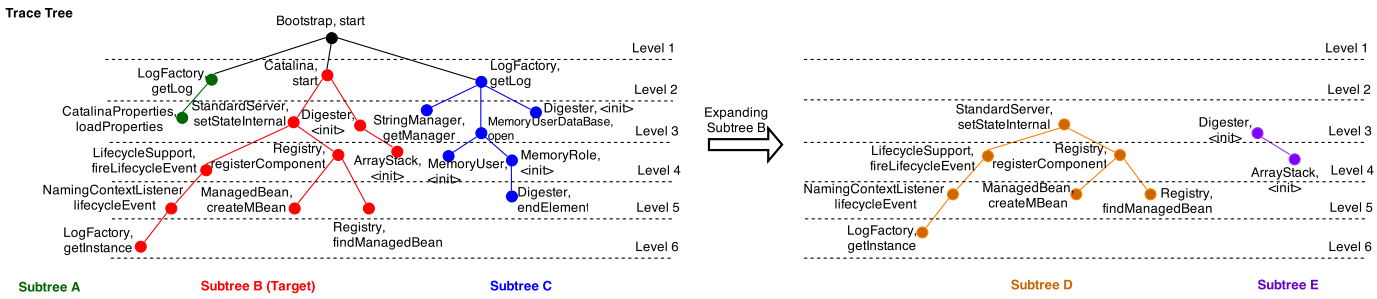

Figure 1. Extracting code elements (subtrees) from execution traces.

subtree) has root *StandardServer.setStateInternal* with size 7, and the subtree *E* (purple subtree) has root *Digester.<init>*, with size 2. The subtree *D* and *E* have roots that are on level 3 of the trace tree, so the extraction process ends with four selected subtrees (A, C, D and E).

3) ***Filtering Identical Subtrees****:* Subtrees that are identical to each other are filtered out to reduce the amount of information to be analyzed. This situation occurs because the loop-removal algorithm may not remove all possible loops. When the discarding process of the identical subtrees is finished, we have a limited and less complex set of subtrees that are organized in terms of features and its threads according to the execution scenario.

C. Phase 3- Classifying Trace Subtrees

The previous process is expected significantly to reduce the number of method calls to be analyzed. However, in some cases, this number is still relatively high, due to the fact that either the number of features or the complexity of the target system are high. So, we propose a classification mechanism based on a set of observable attributes of the remaining subtrees aiming at selecting the most relevant subtrees representing the key architecturally relevant classes.

1) ***Defining the Attributes****:* In order to obtain an accurate classification in supervised learning, it is important to choose relevant attributes to filter desired subtrees. To construct the classifier, we manually analyzed the structure of the resulting subtrees set returned in the previous phase and as well as extract five numeric attributes to perform the classification:

Size of subtrees: Larger subtrees are more likely to provide or consume different services, and so, it would be more inclined to represent a component. We define this attribute as a boolean value, which indicates if the subtree size is larger than the mean size of the system's subtrees. In particular, the attribute related to size, seems to be most discriminative as the others could be somewhat dependent on it.

Distance of the subtree to the main root: subtrees with the root near to the root of the original tree tend to represent more higher-level abstractions.

Number of distinct packages: a subtree with high package variability would confirm the notion that components are not strongly adherent to the package-based structure.

Number of distinct classes: a subtree that contains many distinct classes, encapsulates different responsibilities.

Number of distinct methods: the presence of distinct methods, in the same way as distinct classes, in a subtree could be a sign of coarse-grain responsibility.

2) ***Classifying Subtrees****:* We aim at classifying subtrees into two categories: key and non-key candidates. Even if this classification process could have been applied in earlier stages of the approach, it seemed more coherent to apply it after the removal of redundant calls, because grounded on our hypothesis, we are interested in the variability of packages, classes and methods of a tree and not only in the absolute size where repetitive calls would be considered noise in the classification.

For the training data, we calculate average size of subtrees and then we determine that a subtree should belong to the class of valid subtrees if the granularity was upper than or equal to the average granularity obtained. For other attributes we did not establish rules, because in general subtrees with several nodes have a lot calls from distinct packages, classes and methods. We used a Naïve Bayes classifier as it exhibited higher accuracy (around 94.5% with the training instances) compared to a Neural Network classifier [9]. For each subject system, we extract the attribute values of all subtrees. For the training data, we generated two distinct training groups in order to test two open source systems. To test 9 instances of Ant we considered 377 training instances of JMeter. Finally, to test 377 instances of JMeter, we used 9 training instances of Ant.

3) ***Selecting the Key Architecturally Relevant Classes based on Level-Analysis****:* The final process is to select the key classes from the key subtrees. Although, the subtree root is a good candidate for a key class, there might be other key classes in a subtree, depending on the interest of the developer in understanding the architecture with more or less details. We defined an algorithm that has a cumulative characteristic, i.e., for each covered level of the subtrees, it increases the number of recovered roots (roots and subroots). One question associated with the proposed technique is how to determine a target of n key classes that the approach needs to retrieve. In a real comprehension activity, developers do not know the best value of n, because indeed there is no best value, as it depends on how much detail the developer wants to comprehend. Typically,

they would want to begin with less detail (less classes) and then increase the number of classes as the comprehension process evolves. So, it is reasonable that our approach can rely on an input parameter n indicating the target number of classes. We defined that the number of roots found by the algorithm has to be minimally larger or equal to the number of architecturally relevant classes (n) that the developer expects to find. In [5], the authors also use this kind parameter and return as a result k% of the ranked classes.

III. Preliminary Evaluation

In this section we present the subject systems used to evaluate Keecle and their respective execution scenarios to extract execution traces. The ground-truth key classes, the two Java open source systems and the set of execution scenarios considered in this evaluation were retrieved from [5]. Our approach receives the target number of classes to be recovered as the number of those classes defined in the ground-truth. In [5], they have chosen to retrieve 15% of the classes because the ground-truth had around 10% of the total classes.

JMeter[1] *2.0.1:* is a Java application designed to load test functional behavior and measure performance with approximately 22.234 KLOC. For JMeter, the execution scenario was the same as that used in [5], that is testing a HTTP (HyperText Transfer Protocol) connection for an arbitrary site.

Ant[2] *1.6.1:* is a Java library and command-line tool with approximately 98.681 KLOC, whose functionality is to drive processes described in build files with targets and extension points that dependent on each other. The main usage of Ant is to build Java applications. The execution scenario was the same as that used in [5], that is to build Ant itself.

We present a quantitative evaluation for Keecle based on the values of precision and recall. In Table I and II, we describe how much the six phases reduce the call tree. Due to the fact that each phase filters trees, recall may be hindered.

Table I. Description of the Reduction Phases.

Phase	Description
Ph1	total number of method calls in the traces file for each thread.
Ph2-1	total number of method calls in the traces file for each thread after the compression process.
Ph2-2	total number of key candidate subtrees obtained.
Ph2-3	removing trace subtrees whose content is identical.
Ph3-1	total number of subtrees obtained by classifying processes of the traces.
Ph3-2	number of classes recovered (after possible expansion).

Considering the effort and processing time to apply the approach, we could observe it is lightweight because it takes just a few minutes to complete the task. The phase *Ph1* demands more time as it is related to the system configuration, definition of execution scenarios and trace extraction. After traces are collected, each phase runs in less than a minute on a laptop.

Table II. Reduction of Number of Calls by Phase.

Software	Ph1	Ph2-1	Ph2-2	Ph2-3	Ph3-1	Ph3-2
Ant	1,357,211	624,706	15	9	4	14
JMeter	192,140	73,404	2,301	377	40	18

[1] http://jmeter.apache.org/

[2] http://ant.apache.org/

On Table III (*P=Precision* and *R=Recall*) we show for each phase its impact on the recall reduction, which is mostly unavoidable for improving the precision. One notes that for Ant in phase Ph1 recall of 100% was obtained. Table IV shows the list of recovered classes by Keecle (*C=Correct?*) and the set of missed key classes during the performance keecle.

Table III. Recall and Precision for Phases of the Approach.

Software	Ph1		Ph2-1		Ph2-2		Ph2-3		Ph3-1		Ph3-2	
	R	P	R	P	R	P	R	P	R	P	R	P
Ant	100%	7%	100%	7%	100%	23%	100%	23%	80%	21%	80%	57%
JMeter	93%	2%	93%	2%	93%	2%	93%	2%	93%	3%	93%	72%

An important concept to be mention concerns abstract classes and interfaces. The execution traces capture methods that were effectively called, and which are connected to an object. The class that created this object should not be an abstract class or interface. In this context, if we have in the documentation an abstract class or interface as a key class and during the capture of traces, a concrete class that extends or implements these situations was shown, so we will consider abstract classes and interfaces in the our results.

For JMeter, *JMeterEngine* is a interface. Execution traces captured *StandardJMeterEngine*, a concrete class that implements the *JMeterEngine*. So, the *JMeterEngine* was considered in our results. The *JMeterGuiComponent* was not captured as it is an interface and the traces did not capture a class to implement it. To perform the phase Ph3-2, our input parameter corresponds to 14 key classes and 40 key subtrees recovered in the phase Ph3-1. In this context, on the first level 18 roots were recovered (classes) as identical roots existed between them. Thus, the recall and precision values were 93% and 72%, respectively the value of the F-measure was equal to 81% as shown in Table V.

For Ant, the execution traces captured *Task*, a concrete class that implements the *TaskContainer*. In phase Ph3-2, our input parameter corresponds to 10 key classes and 4 key subtrees recovered in phase Ph3-1. In this context, the research was performed on the next two levels and recovered a total of 14 classes. The *Main* and *Project* classes were missed in phase Ph3-1. Thus, the recall and precision values were 80% and 57%, respectively the value of the F-measure was equal to 66% as shown in Table V.

Table IV. List of Recovered Classes for Keecle.

Ant		JMeter	
Recovered Classes	**C**	**Recovered Classes**	**C**
ant.ProjectHelper	yes	threads.ThreadGroup	yes
ant.Target	yes	threads.JMeterThread	yes
ant.UnknownElement	yes	threads.TestCompiler	yes
ant.Task	yes	jorphan.logging.LoggingManager	no
ant.TaskContainer	yes	samplers.Sampler	yes
helper.ProjectHelper2$ ElementHandler	yes	gui.action.AbstractAction	yes
ant.ComponentHelper	no	engine.PreCompiler	yes
ant.IntrospectionHelper	yes	protocol.http.sampler.HTTPAbstractImpl$	no
ant.RuntimeConfigurable	yes	CachedResourceMode	
ant.AntTypeDefinition	no	testelement.TestListener	yes
ant.TaskAdapter	no	jmeter.JMeter	no
taskdefs.Javac	no	engine.JMeterEngine	yes
helper.ProjectHelper2$ProjectHandler	no	samplers.SampleResult	yes
-	-	control.gui.TestPlanGui	yes
-	-	gui.tree.JMeterTreeModel	yes
-	-	testelement.TestElement	yes
-	-	sampler.HTTPSamplerFactory	no
-	-	testelement.TestPlan	yes
Missed Key Classes of [5]			
Ant		**JMeter**	
ant.Project		gui.JMeterGuiComponent	
ant.Main		-	

Table V shows the results for Keecle and Zaidman-Demeyer's approach [5], indicating the values of precision, recall and the F-measure obtained for the two systems. For Ant and JMeter we preferred to use the values reported in their paper. One notes that the F-measure values of Keecle outperformed the results in [5].

Finally, Table VI shows the average for the attributes of each subtree for the applications. One notes that, in general, the subtrees are formed by method calls from different packages and classes which reinforces the notion that components seem to span different packages.

Table V. RECALL AND PRECISION FOR THE PHASE PH3-2 AND [5]'S APPROACH.

Software	# Classes Doc	# Classes Recovered		Recall (%)		Precision(%)		F-Measure(%)	
		Keecle	[5]	Keecle	[5]	Keecle	[5]	Keecle	[5]
Ant	10	14	19	80%	90%	57%	47%	66%	62%
JMeter	14	18	28	93%	93%	72%	46%	81%	62%

Table VI. AVERAGE ATTRIBUTES IN EACH SUBTREE.

Software	Size	Root Level	# Packages	# Classes	# Methods
Ant	849,5	7,25	4,75	15	31,25
JMeter	168,725	6,05	9,3	19,525	36,75

IV. RELATED WORK

Several techniques are proposed in literature to extract software architecture. A recent study [10] performed an evaluation with six software architecture recovery techniques based on clustering. The three most relevant we review in the following. ACDC [11] recovers components discovering clusters that follow patterns commonly observed in decompositions of large systems. It automatically assigns meaningful names to the clusters. In some cases, the tool can produce very small clusters. Keecle criteria benefit larger subtrees, and the limitation occurs when small relevant subtrees are discarded. Fortunately, these situations seem to be exceptions.

WCA [12] is a clustering algorithm based on the inter-cluster distance during the clustering process. Experiments with WCA [10] showed that for some inputs, the MoJoFM metric was not calculated because of memory errors. We did not evaluate Keecle for the MoJoFM metric, but for the cases presented, precision and recall were adequately calculated.

In [13], a technique was designed to recover concerns associated with components to facilitate an understanding of the cluster meaning. Another kind of elements recovered by the approach is that of connectors, which describe the interaction between the components. The evaluation with eight systems produced reasonable results.

Zaidman and Demeyer's work is the most closely related to ours [5]. They proposed a technique to automatically identify so-called key classes of a software system that can be useful for a software engineer who is trying to get a high-level overview of a system that he is unfamiliar with. Their technique is based on the identification of tightly coupled classes. They also take into account indirect coupling through the application of a webmining algorithm. In our work, software components are formed by large trace subtrees containing key classes.

V. CONCLUSION

In this paper, we have proposed an approach based on mining execution trace method calls to capture architecturally relevant classes. Several phases were proposed to improve precision and recall.

One of our goals was to show that we can deal effectively with the volume of data traces, using compression techniques and through the removal of irrelevant data. A study using two target systems, showed that the approach produced encouraging values of recall and precision outperforming previous work in the literature.

As future work there are some opportunities to be explored, such as hybridizing the approach with other architecture reconstruction techniques, especially in cases where dynamic data is not complete. Furthermore, conducting a controlled experiment using a larger number of systems, within different domains and implemented in different languages is necessary to further evaluate the generality of the approach.

ACKNOWLEDGMENT

We would like to thank the Brazilian agencies FAPEG, FAPEMIG, CAPES and CNPq (grant 201410267000025 and grant 475519/2012-4) for partially funding this research.

REFERENCES

[1] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining ground-truth software architectures," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 901–910. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486911

[2] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[3] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Trans. Software Eng.*, vol. 35, no. 4, pp. 573–591, 2009.

[4] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, ser. IWPC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 194–.

[5] A. Zaidman and S. Demeyer, "Automatic identification of key classes in a software system using webmining techniques," *J. Softw. Maint. Evol.*, vol. 20, no. 6, pp. 387–417, Nov. 2008. [Online]. Available: http://dx.doi.org/10.1002/smr.v20:6

[6] L. Tahvildar and K. Kontogiannis, "Improving design quality using meta-pattern transformations: A metric-based approach: Research articles," *J. Softw. Maint. Evol.*, vol. 16, no. 4-5, pp. 331–361, Jul. 2004.

[7] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Trans. Softw. Eng.*, vol. 29, no. 3, pp. 210–224, Mar. 2003.

[8] A. Hamou-Lhadj, T. Lethbridge, and L. Fu, "Challenges and requirements for an effective trace exploration tool," in *IWPC*, 2004, pp. 70–78.

[9] R. O. Duda, P. E. Hart, D. G. Stork, C. R. O. Duda, P. E. Hart, and D. G. Stork, "Pattern classification, 2nd ed," 2001.

[10] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques." in *ASE*. IEEE, 2013, pp. 486–496.

[11] V. Tzerpos and R. C. Holt, "Acdc : An algorithm for comprehension-driven clustering," in *In Proceedings of the Seventh Working Conference on Reverse Engineering*. IEEE, 2000, pp. 258–267.

[12] O. Maqbool and H. A. Babri, "The weighted combined algorithm: A linkage algorithm for software clustering." in *CSMR*. IEEE Computer Society, 2004, pp. 15–24.

[13] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 552–555.