TOOL SUPPORT FOR CAPTURING THE ESSENCE OF A CONCERN IN SOURCE CODE

Chuntao Fu
University of Nebraska at Omaha

Follow this and additional works at: https://digitalcommons.unomaha.edu/studentwork

Part of the Computer Sciences Commons

Recommended Citation
https://digitalcommons.unomaha.edu/studentwork/2914

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.
TOOL SUPPORT FOR CAPTURING THE ESSENCE OF A CONCERN IN SOURCE CODE

A Thesis
Presented to the
Department of Computer Science
and the
Faculty of the Graduate College
University of Nebraska
In Partial Fulfilment
of the Requirements for the Degree
Master of Science in Computer Science
University of Nebraska at Omaha
by
Chuntao Fu
Omaha, Nebraska
August, 2017

Supervisory Committee:
Harvey Siy, Ph.D.
Mahadevan Subramaniam, Ph.D.
Myoungkyu Song, Ph.D.
William Mahoney, Ph.D.
Software evolves constantly to adapt to changing user needs. As it evolves, it becomes progressively harder to understand due to accumulation of code changes, increasing code size, and the introduction of complex code dependencies. As a result, it becomes harder to maintain, exposing the software to potential bugs and degradation of code quality. High maintenance costs and diminished opportunities for software reusability and portability lead to reduced return on investment, increasing the likelihood of the software product being discarded or replaced. Nevertheless, we believe that there is value in legacy software due to the amount of intellectual efforts that have been invested in it. To extend its value, we utilize the common practice of identifying the pieces of code relevant to a given concern. Identifying relevant code is a manual process and relies on domain and code expertise. This makes it difficult to scale to large and complex code. In this thesis, we propose several automated approaches for capturing the essential code that represents a concern of interest. We utilize dynamic program analysis of execution traces to identify a relevant code subset. Information retrieval techniques are then utilized to improve the accuracy of the capture, refine the process, and verify the results.
ACKNOWLEDGMENTS

I would like to express my deepest gratitude for my thesis advisor Dr. Harvey Siy for his mentorship and excellent guidance in finishing this thesis research. I also thank the members of my committee for their support throughout the research process and their review and feedback on my thesis. Specifically I thank Dr. Mahadevan Subramaniam for his advise on dominators and information retrieval analysis, Dr. Myoungkyu Song for his help with static analysis tools, and Dr. William Mahoney for his advice on thesis writing. I also want to thank my family and friends for their support. I would not have done it otherwise.
Contents

Contents iv

List of Figures vii

List of Tables viii

1 Introduction 1

1.1 Software Concern Definition 2

1.2 Software Concern Isolation 3

1.3 Organization 4

2 Literature Review 5

2.1 Static-Based Program analysis 6

2.2 Dynamic-Based Program Analysis 8

2.3 IR-Based Program Analysis 9

2.4 Other Approaches 10

3 Methodology 11

3.1 Relevant Code Subset Identification 11

3.1.1 Identifying Program Entry Points 12

3.1.2 Static Call Graph Generation 12
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.3</td>
<td>Designing Program Execution Scenarios</td>
<td>14</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Dynamic Call Graph Generation</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Relevant Code Subset Minimization</td>
<td>18</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Frequency-Based Code Analysis</td>
<td>18</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Algorithm-Based Code Analysis</td>
<td>19</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Information-Retrieval-Based Code Analysis</td>
<td>20</td>
</tr>
<tr>
<td>3.2.3.1</td>
<td>Latent Dirichlet Allocation</td>
<td>24</td>
</tr>
<tr>
<td>3.2.3.2</td>
<td>Latent Semantic Indexing</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Validation Strategy</td>
<td>25</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Use of Independent Techniques</td>
<td>25</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Tool Validation</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Summary and Discussion</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>Case Study Results</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>Systems Under Study</td>
<td>30</td>
</tr>
<tr>
<td>4.1.1</td>
<td>JHotDraw</td>
<td>31</td>
</tr>
<tr>
<td>4.1.2</td>
<td>ArgoUML</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Analysis Results</td>
<td>31</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Dynamic Call Graph Tools</td>
<td>32</td>
</tr>
<tr>
<td>4.2.1.1</td>
<td>Javashot</td>
<td>32</td>
</tr>
<tr>
<td>4.2.1.2</td>
<td>Java-call-graph</td>
<td>33</td>
</tr>
<tr>
<td>4.2.2</td>
<td>JHotDraw Analysis</td>
<td>33</td>
</tr>
<tr>
<td>4.2.2.1</td>
<td>Program Entry Points</td>
<td>33</td>
</tr>
<tr>
<td>4.2.2.2</td>
<td>Program Execution Scenarios</td>
<td>34</td>
</tr>
<tr>
<td>4.2.2.3</td>
<td>Frequency Analysis Result</td>
<td>34</td>
</tr>
<tr>
<td>4.2.2.4</td>
<td>Algorithm Analysis Result</td>
<td>36</td>
</tr>
</tbody>
</table>
4.2.2.5 Information Retrieval Analysis Result . . . . . . 37
4.2.3 ArgoUML Analysis . . . . . . . . . . . . . . . . . . . . . . . . . 38
  4.2.3.1 Program Entry Points . . . . . . . . . . . . . . . . 39
  4.2.3.2 Program Execution Scenarios . . . . . . . . . . . . 39
  4.2.3.3 Frequency Analysis Result . . . . . . . . . . . . . . 41
  4.2.3.4 Algorithm Analysis Result . . . . . . . . . . . . . . 41
  4.2.3.5 Information Retrieval Analysis Result . . . . . . . 44
4.3 Discussion and Limitations . . . . . . . . . . . . . . . . . . . 46

5 Conclusion and Future Works 49
  5.1 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
  5.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

Bibliography 52
List of Figures

3.1 Static Call Graph ......................................................... 14
3.2 Dynamic Call Graph ..................................................... 17
3.3 Dominator Call Graph .................................................... 20
3.4 Dominator Tree .......................................................... 21
3.5 Dynamic Program Analysis & IR Overview ............................ 22
3.6 IR Coupus Creation ....................................................... 23
3.7 Analysis Work Flow ...................................................... 27
4.1 JHotDraw Undo Concern Call Graph ................................. 35
4.2 JHotDraw Undo Concern Participants in Canfora and Cerulo [7] 35
4.3 Dominator Tree for JHotDraw Undo Based on Frequency Analysis. The colored nodes represent classes present in all execution traces collected. 36
4.4 Use Case Diagram Scenario ............................................. 39
4.5 Class Diagram Scenario ................................................. 40
4.6 State Diagram Scenario .................................................. 40
4.7 Sequence Diagram Scenario ............................................ 41
4.8 Dominator Tree for ArgoUML Based on Frequency Analysis .... 43
List of Tables

2.1 Literature Table Summary .................................. 6

4.1 LDA Analysis Ouput For JHotDraw. The number next to each word
    indicates how likely each word is related to its topic. .................. 37

4.2 LSI Partial Analysis Ouput For JHotDraw .......................... 38

4.3 Partial Frequency Analysis Output For State Diagram Scenario ...... 42

4.4 LDA Analysis Output For State Diagram Scenario. The number next to
    each word indicates how likely each word is related to its topic. ....... 45

4.5 LSI Partial Analysis Ouput For State Diagram Scenario ................. 46
Chapter 1

Introduction

Software development is an evolving process, it evolves constantly to adapt to changing needs. This ongoing process can be referred as an incremental operation, because newer features or functionality are always added on top of the existing software system. As it evolves, it becomes progress harder to understand due to accumulation of code changes, increasing code size, and the introduction of complex code dependencies. In other words, as we add more functionality to the software system, there are increasing chances that the software will be exposed to potential bugs and degradation in code quality, increasing the potential risks to the software. Due to the lack of software understanding, these problems will not be detected easily. They will be carried over to the next version whenever the software evolves. Eventually, making new changes becomes extremely difficult, and the software demands more attention in maintenance, driving up the costs of usage every time. Due to high maintenance costs and diminished opportunities for software readability and portability lead to significant reduced return on investment, shortening the life span of the software, increasing the likelihood of the software product being discarded or replaced. Nevertheless, we believe that there is still value in these legacy software due to the amount
of intellectual efforts that have been invested in it. Therefore our goal is to extent its value. Source code localization has been a popular yet challenge topic in software engineering. One common practice is to identify the pieces of code relevant to a problem under investigation. To extend its value, we utilize this common practice to identify a code subset that plays the most central roles in a given software concern, making it convenient for potential future reuses.

1.1 Software Concern Definition

We observe that a software system is a combination of multiple complex software concerns. A concern is considered to be any conceptual software unit that is considered valuable to a stakeholder, i.e. a feature or a functionality. For example, the hot key combination **CTRL/Command – Z** in a common text editor is used to undo the last change made to the file you are editing. The undo feature here can be considered a concern, which is implemented at the back probably using a Stack or a Queue data structure, which involving a set of classes and methods. In another words, a software concern maps to a subset of code that performs the services the concern is representing in the software. We hypothesize that if we can isolate such concerns in the code, those code subsets captured would be easier to understand, and possibility reuse, thus extending the value of those software that are facing the decisions of being redeveloped or replaced. By understanding a concern, we can identify the classes or modules that play the most central roles in that concern, gain better knowledge of the code relationship and understand how these code modules working together. As a result it help to facilitate the overall understanding of the entire system.
1.2 Software Concern Isolation

Identifying relevant code to a problem often times is a manual process and relies on domain and code expertise. This makes it difficult to scale to large and complex code. Prior attempts to isolate concerns for code identification focus on static program analysis. FEAT [22] is an Eclipse plugin developed under static program analysis for describing, locating and analyzing concerns in source code. It requires extensive developer skills and does not perform any code subset minimization. In this thesis, we propose several automated approaches for capturing the essential code that represents a concern of interest. We utilize dynamic program analysis by exploring execution traces as our analysis foundation. We introduce the concept of program entry point and program execution scenarios to guide the dynamic program execution to allow us quickly get to the relative portion of the program. The dynamic call traces captured in this case are directly related to the given software concern. By constructing dynamic call graphs with program execution traces, we are able to force the analysis to start from the heart of the problem. We also introduce several code minimization approaches to help us achieve the level of capture to the granularity that most related to the concern under investigation. There are frequency-base, algorithm-based, and information retrieval based analysis. Frequency-based analysis is to used to identify those classes that appear most frequently across all execution scenarios and their names best match the given concern context. Algorithm-base analysis leverages graph algorithm for call graph minimization. It can be used as independent technique as well as an analysis upon frequency-based analysis to reduce the negative impacts cause by name matching. Information retrieval techniques are utilized to improve the accuracy of the capture, refine the process, and verify the results.
1.3 Organization

This thesis organize into 5 chapters, with chapter 1 and chapter 2 being the introduction and related literature review section. The chapter 3 explores the details of the methodology behind concern capturing, and chapter 4 shows the case study results of applying the methodology to some real world open source projects. The conclusion is in chapter 5, where we will summarize the thesis research and talk about some future works.
Chapter 2

Literature Review

Mapping high level concerns to source code has been studied extensively due to its applications in software evolution. We observe two common objectives in identifying concerns. The first is locating a specific concern with respect to a task. The second is identifying all concerns in code for purposes of understanding an unfamiliar program or restructuring a legacy program.

Our literature search on concern location included research on feature location, where features are concerns related to program behavior that are triggered by user and exhibit observable behavior [9], and aspect mining, where aspects are crosscutting concerns [11] or recurring features whose code is scattered across multiple modules.

We present a broad range of papers regarding various types of program analysis techniques, as shown in Table 2.1. Regardless of the analysis purpose of each approach, we can divide them into 4 major categories: static-base program analysis, dynamic-based program analysis, information-retrieval-based analysis and other techniques.
<table>
<thead>
<tr>
<th>No.</th>
<th>Objective</th>
<th>Techniques</th>
<th>Systems Studied</th>
</tr>
</thead>
<tbody>
<tr>
<td>[22]</td>
<td>build a concern graph during code investigation</td>
<td>static analysis</td>
<td>JHotDraw (save), Jex, Redback, JEdit autosave, ArgoUML (annotation)</td>
</tr>
<tr>
<td>[17]</td>
<td>finding all aspects through high fan-in</td>
<td>static analysis</td>
<td>PetStore, JHotDraw, Tomcat</td>
</tr>
<tr>
<td>[9]</td>
<td>locating all features</td>
<td>static, dynamic analysis, FCA</td>
<td>Mosaic, Chimera (history &amp; bookmarks), Agile tester platform</td>
</tr>
<tr>
<td>[14]</td>
<td>mining all aspects</td>
<td>static analysis</td>
<td>JHotDraw</td>
</tr>
<tr>
<td>[21]</td>
<td>locating code relevant to modification task</td>
<td>static analysis</td>
<td>JEdit (autosave), Azureus, BitTorrent</td>
</tr>
<tr>
<td>[8]</td>
<td>concept location for change request</td>
<td>static analysis</td>
<td>Mosaic (media files management)</td>
</tr>
<tr>
<td>[25]</td>
<td>mining all aspects</td>
<td>dynamic analysis, FCA</td>
<td>internal</td>
</tr>
<tr>
<td>[3]</td>
<td>feature identification</td>
<td>static, dynamic analysis</td>
<td>Mozilla (save bookmark)</td>
</tr>
<tr>
<td>[4]</td>
<td>feature identification</td>
<td>static, dynamic analysis</td>
<td>Mozilla, Firefox, Chimera, ICEBrowser (save bookmark), JHotDraw, XFig (draw circle)</td>
</tr>
<tr>
<td>[15]</td>
<td>discover linguistic topics</td>
<td>LSI, Distribution Map</td>
<td>jEdit, JBoss</td>
</tr>
<tr>
<td>[16]</td>
<td>map concepts to relevant source code</td>
<td>LSI</td>
<td>NCSA Mosaic (font properties)</td>
</tr>
<tr>
<td>[27]</td>
<td>reveal connection between features and source code</td>
<td>static analysis, Vector Space Model</td>
<td>GNU DC, GNU UnRTF</td>
</tr>
<tr>
<td>[19]</td>
<td>feature location in source code</td>
<td>dynamic analysis, LSI</td>
<td>Mozilla (feature identification, bug location)</td>
</tr>
<tr>
<td>[26]</td>
<td>separate crosscutting concern from core elements</td>
<td>static analysis, PageRank</td>
<td>ORBacus, JHotDraw, AspectJ, WebSphere AS, PADRES, Prevayler, hSQL, Derby</td>
</tr>
<tr>
<td>[10]</td>
<td>find features</td>
<td>dynamic analysis, ranking</td>
<td>HTMLUnit, Axion (bug fixes)</td>
</tr>
<tr>
<td>[23]</td>
<td>locate action oriented concerns</td>
<td>NLP</td>
<td>JBidWatcher, JavaHMO, Jajuk, iReport</td>
</tr>
<tr>
<td>[24]</td>
<td>discover crosscutting concern</td>
<td>NLP</td>
<td>PetStore (customer notification)</td>
</tr>
<tr>
<td>[1]</td>
<td>mining all aspects</td>
<td>commit analysis</td>
<td>PostgreSQL, NetBSD</td>
</tr>
</tbody>
</table>

Table 2.1: Literature Table Summary

### 2.1 Static-Based Program analysis

Static analysis techniques make use of information available by reading and analyzing syntactic elements in source code.
Robillard and Murphy [22] propose a tool, FEAT, which developers can use for interactive code exploration while identifying concerns related to modification tasks. FEAT tracks and relates identified concerns and uses static calling dependencies to guide users to inspect related code. Robillard [21] extend static analysis further by identifying concerns based on an analysis of the topology of structural dependencies in a program. The method takes as input a set of program elements of interest to a developer and produces a fuzzy set describing other elements of potential interest.

Several static analysis techniques are proposed for identifying all concerns in code, particularly, crosscutting concerns. Static techniques take advantage of repetitive syntactic patterns in the source code to identify likely crosscutting concerns. Marin, et al. [17] use fan-in data from analyzing static method call dependencies to identify crosscutting concerns, with the assumption that crosscutting concerns have methods that are called from many different places. Krinke [14] proposed a different approach for identifying crosscutting concerns that examines the control flow graph searching for recurring patterns of method calls, indicating the recurring functionality often associated with aspects. Ranking heuristics have also been used to identify the most likely concerns. For example, Zhang and Jacobsen [26] used a random walk to compute a variation of Google’s PageRank heuristic [18] to use in distinguishing core from crosscutting concerns. The heuristics compute popularity based on the number of direct and indirect references to it (an element is popular when frequently visited from different elements, and is likely to be crosscutting), and significance based on number of elements it references directly or indirectly (an element is significant if it references a large number of distinct elements, and is less likely to be crosscutting).
2.2 Dynamic-Based Program Analysis

Dynamic analysis techniques make use of data collected from running the program with respect to a set of execution scenarios. Such data include call traces, execution profiles, and statement-level execution traces. The execution scenario is typically related to the concern of interest. Dynamic data precisely show what parts of the program are actually involved in the scenario.

Antoniol, et al. [4] conduct an epidemiological analysis of the execution traces to aid in locating a feature of interest. This type of analysis is inspired by epidemiology where population data is analyzed to separate individuals with a disease from those that do not. Similarly, execution traces are analyzed to isolate events associated with a feature of interest from events that are not related. The approach identifies events that are more frequent for scenarios in which a feature of interest is exercised.

Eisenbarth, et al. [9] propose a method for locating all features in source code through using dynamic call trace data to map features to code through formal concept analysis. The resulting mapping is then used to inform manual inspection of the static dependency graph. Tonella and Ceccato [25] use a similar technique with formal concept analysis to mine aspects.

Ranking heuristics have also been used with dynamic data, especially where there is a very large data set to analyze. For example, Eisenberg and De Volder [10] proposed a technique that collects execution data from running comprehensive test suites of software applications. The technique ranks the methods most closely associated to a given feature by using three heuristics, multiplicity (based on frequency of a method being exercised in a test set), specialization (degree of a method being exercised exclusively for a given test set), and depth (average call depth of a method, with the assumption that methods closely related to a feature will be more directly exercised.
2.3 IR-Based Program Analysis

Traditional Information Retrieval (IR) techniques mainly solve the problems for finding highly correlated documents based on a given query in a code and comments collection. An IR system generally provides an ordered list of documents with similarities from a given searching query. The document source creates a corpus that serves as the basic foundation for IR analysis. The source typically derives from source codes and any other relative forms of documents or descriptions. Nowadays, there have been a growing number of applications of IR system to solve software engineering problems supporting aspects of software maintenance and evolution tasks.

Lo Kwun Kit et al. [13], use Latent Semantic Analysis (LSA) for aspect-oriented requirements analysis to identify concerns that behaviorally influence other concerns. LSA helps identify useful concern clusters, and helps reduce the number of falsely identified aspectual requirements.

Semantic Clustering [15], developed by Adrian Kuhn, et al., is a technique based on the use of Latent Semantic Indexing (LSI) and clustering to group source artifacts that use similar vocabulary in order to exploit linguistic information i.e. identifier names and comments to discover the semantic topics of a system. The linguistic topics discovered can be visualized with help of a Distribution Map.

Feature location using IR has always been a popular topic in software engineering activities. Andrian Marcus et al. [16] leverages LSI to map concepts expressed in natural languages by the programmers to the relevant parts of the source code. LSI is used to accept two sets of query searching: 1) user specified query; 2) use of automated generated queries, queries contain both words and identifiers from the source code.

There also have been works in feature location by combining IR with non-IR tech-
niques to improve the analysis. Wei Zhao and Lu Zhang[27] combine static program analysis with Vector Space Model (VSM) for feature location. VSM acquires the initial connection between a feature and function, Branch-Reversing Call Graph then is used to recover both relevant and specific computational units for each feature. Denys Poshyvanyk et al.[19] combine dynamic program analysis with LSI for feature identification. Two sets of execution scenarios are used for Scenario Based Probabilistic (SBP) event ranking, with one set exercising a functionality of interest and the other set being irrelevant to the functionality. Combining SBP and LSI ranking helps to improve the precision of feature identification.

2.4 Other Approaches

Other approaches for identifying concerns leverage additional information available with code. Code comments and consistent identifier naming conventions make code artifacts amenable for natural language processing. For example Shepherd, et al. [23] make use of natural language processing to identify action-oriented concerns. Specifically, part-of-speech tagging is used to extract verb-object pairs from identifiers and code comments; these in turn are traced back to their uses, with modules using common action-oriented identifiers considered to be implementing the same concern.

The common use of version control repositories to track code changes also makes it possible to analyze patterns of developer activities through code histories. For example, Adams, et al. [1] identify major concerns in large systems by analyzing the source code history to statistically cluster functions, variables, types and macros that have been changed together intentionally.
Chapter 3

Methodology

This chapter introduces the overall methodology for capturing the essence of concern in source code, which can be divided into three major tasks: code identification, code minimization and code validation. The chapter steps into the details of each task, elaborating on various sub-tasks that need to be performed, the motivations and relationship among all the tasks.

3.1 Relevant Code Subset Identification

Source code identification is the first step in concern capturing. The major challenge is that a concern exists in a semantic context, it is very difficult to define universal rules that work for all cases. Thus, semantic relevance is a relative concept, which heavily replies on the specific operational context, i.e. a specific functionality or feature in a complex software system.
3.1.1 Identifying Program Entry Points

A program entry point defines the specific operational context for an identified software concern. It is a starting point in a software that can be executed to bring out the potential relevant components to the concern that we are currently investigating. It also serves as the start capturing point for a well-designed scenario (will be explained in section 3.1.3) during a program execution.

In general, the program entry points as the concern start capturing points have the following characteristics:

- They are the class names or method names in source code that will firstly be used to perform a search operation during the concern investigation.

- They are actionable keywords or synonym that best describe the identified software concern, e.g. for capturing a save concern, the actionable key words could have “action save”, “save”, “restore” or “persist”.

- They are part of a software component code implementation, which can be located through program execution.

A software concern can potentially map to multiple program entry points. Generally, the more program entry points we use, the more complete the code subset captured will be, and the more accurate those entry points are, the more relevant the capturing results will be.

3.1.2 Static Call Graph Generation

Searching for program entry points through manual code investigation is very time consuming, and often times require extensive developer skills especially when the software is complex and the code size is large. To reduce these intensive inefficient
searching activities, we obtain program entry points by constructing static call graphs via static program analysis.

Static program analysis is the analysis of a piece of computer software that is performed without necessary executing any programs. It performs the analysis by targeting some version of the source code of that piece of software. Static program analysis offers a comprehensive as well as flexible source code behaviour analysis from an individual statement level to the entire code base. More importantly, this type of source code investigation process is usually assisted with automated tools to improve the efficiency and accurateness.

There are various tools available for static program analysis as we have introduced in the literature review section. In this thesis, we chose IBM’s Watson Libraries for Analysis (WALA) to be our static program analysis tool to construct static call graphs. A static call graph is the dependency graph obtained by exploring the implicit and explicit calling dependencies between classes and methods.

**Listing 3.1: Sample Code for Static Call Graph Extraction**

```java
1 class A {}
2 class B extends A {
3     void m1() {
4         C c = new C();
5         c.m2();
6     }
7 }
8 class C {
9     void m2() {}
10 }
```

In Listing 3.1, class B has an explicit calling dependency on class C due to the m2 method invocation in m1 in class B. As Figure 3.1 shows, in addition to this,
the static call graph can also capture the implicit dependency between class B and class A as well as the dependency between each class and java.lang.Object generated based on the Java inheritance chain. By leveraging the rich information captured by static call graph, together with name matching techniques, program entry points can be conveniently obtained by customizing searching criteria over method calling dependency traces.

3.1.3 Designing Program Execution Scenarios

In spite of having a good program entry point, a well-designed program execution scenario can help to localize relevant code subsets during a program execution. A program execution scenario is a test scenario designed to exercise the code modules closely related to a given software concern. It helps to guide the process of program execution so that code modules reflecting the concern will be fully captured in the execution trace.
Similar to program entry points, there could be multiple program execution scenarios designed for an identified software concern. There are two major reasons for using multiple execution scenarios:

- Each unique program execution scenario provides a set of unique program execution output containing relevant information for concern capturing. Multiple program execution scenarios can provide more data sets for later execution trace analysis, i.e. frequency analysis, which helps to improve the overall accuracy of class inclusion.

- Programs have limited number of ways to be executed, there will be even less ways to execute a specific software functionality. Theoretically, if we can exhaustively run every possible execution scenario, we can guarantee the completeness of the final concern set captured.

3.1.4 Dynamic Call Graph Generation

While static program analysis is comprehensive, the analysis process is cumbersome due to state explosion and imprecise due to dynamic binding issues [14]. To fully make use of the advantages of program entry points and program execution scenarios, dynamic program analysis appears to be a good choice for code subset localization.

Dynamic program analysis is the analysis of the properties of a running system, which involves the investigation of these properties using the information gathered at run time. With proper program input (program execution scenarios in our case) dynamic program analysis provides a faster and more accurate approach to identify code subset that are relevant to the problem that we are investigating; In addition, it works well with dynamic feature of languages such as Java. Compared to static program analysis, in general, static program analysis is more comprehensive but im-
precise, dynamic program analysis is more accurate but incomplete. However, the incompleteness of dynamic program analysis can be made up by executing the program with adequate set of program execution scenarios.

Similar to static program analysis, we are able to generate dependency call graphs for utilizing the program execution traces collected under each program execution scenario. Unlike static call graph, a dynamic call graph deals with only explicit method calling dependencies.

Listing 3.2: Dynamic Call Graph Source Code

```java
1 class A {}
2 class B extends A {
3     int number = (int)(Math.random() * (10 + 1));
4     void m1() {
5         if (number % 2 == 0) { // if even number
6             D d = new D();
7             d.m3();
8         } else {
9             C c = new C();
10            c.m2();
11        }
12    }
13 }
14 class C {
15     void m2() {}
16 }
17 class D {
18     void m3() {}
19 }
```

To elaborate, let us modify the code segment listed in Listing 3.1 by adding another
class D and a condition statement inside method m2 in class B that invokes method m2 or m3 depends on whether the random number generated is odd or even. Let us also assume that under a designed scenario, the method m1 was executed and the random number generated was 8. Figure 3.2 shows the dynamic call graph generated for the execution.

![Dynamic Call Graph](image)

Figure 3.2: Dynamic Call Graph

As we can see, the calling dependency captured only includes the if part of code that was being executed. The static call graph would have captured the calling dependencies within else part of code too, which does not accurately reflect the program inputs.

To be able to quickly get to the heart of code localization for a given software concern, dynamic program analysis appears to be a better fit to our needs of concern capturing. There also are different tools that can be used for constructing dynamic call graphs to achieve different analysis purpose. We have chosen to investigate Javashot and Java-call-graph based on our analysis needs. See section 4.2.1 for a detailed introduction of these two tools.
3.2 Relevant Code Subset Minimization

Once a relevant part of source code has been identified, one important question to ask is “Are they all source code that we need to identify for this concern?” Minimal achieving problem in computer science is argued to be an undecidable problem, our goal is to get rid of the unnecessary captures in the code subsets captured as much as possible, so that final result set is lean and concise. This leads us to the second major task in the methodology, relevant source code minimization. Three types of minimization approaches, frequency-base, algorithm-base and information-retrieval-base will be explained separately.

3.2.1 Frequency-Based Code Analysis

Frequency not only shows number of times a method has been executed during one program execution scenario, but the overlap degrees a class has for various execution scenarios. Class overlap indicates the common classes executed across multiple execution scenarios. A class has a higher overlap degree if it has a higher frequency counting in execution traces for all the selected scenarios executed. Class and method name matching with concern context is the main criteria to determine how close a class or method is to the concern being captured.

We have define a relation function heuristics to decide whether a class is related to a concern and when a class should be included in the concern set. A class is related to a concern if:

- the class appears frequently in all execution scenarios, and
- the class name or the name of its calling function best matches the concern context, or
• another related class calls it with high frequency

Once we obtain the frequency output for all the selected program execution scenarios, we can construct a dynamic call graph using frequency output as filter to better visualize the frequency analysis output.

### 3.2.2 Algorithm-Based Code Analysis

Another code minimization approach is algorithm based by implementing efficient graph algorithms with the data collected from execution traces. The reason behind this is that the analysis performed by frequency analysis alone is not sufficient to capture all classes that are related to the given software concern. Frequency analysis relies heavily on name matching techniques, which could easily introduce some false positive captures if the naming of classes or methods are bad. And it is rarely the case that the concern relatedness of a class or method can be simply determine by their naming.

To overcome this difficulty, a graph algorithm called dominator algorithm[2], comes into play. We use pre-dominator relationship in our case. In a graph, a node \( D \) dominates a node \( N \) if every path from the entry node to \( N \) must go through \( D \). And by definition, every node dominates itself. With the dominance relationship defined between nodes, we can conveniently construct a dominator tree by implementing the dominator algorithm on program execution traces. We then can search dominators for each one of those classes that identified having high overlapping degrees. This helps to bring in classes that related to the given software concern even if they don’t share the relative name matching properties with the classes identified in frequency analysis.

Figure 3.3 shows a sample call graph. Suppose that the green nodes \( L \) and \( J \) in
the graph indicates these two nodes have high class overlap degree across multiple program execution scenarios. By constructing a dominator tree based on this graph through implementing dominator algorithms, as shown in Figure 3.4, we are able to identify 4 extra nodes as their dominators: node G, node C, node D and node R, which should be included in the final concern set captured. With frequency analysis alone, we would have missed these three nodes.

3.2.3 Information-Retrieval-Based Code Analysis

Previous code minimization approaches are structure based analysis, where the analysis needs to proceed based on some form of structure representation of the problem
i.e. utilize abstract syntax tree for static program analysis, or graph structure for dynamic program analysis. This section introduces a different analysis approach which operates on textual representation of the problem. Information retrieval (IR) techniques have long been applied on text document analysis as well as some other textual format document analysis. Traditional IR techniques mainly solves the problems for finding highly correlated documents based on a given query in a code and comments collection. An IR system generally provides an ordered list of documents based on similarity between collections and given query. Nowadays, there have been a growing number of applications of IR system to solve software engineering problems like feature localization, regression test and traceability links recovery by extracting useful information from the combination of source code and other software documents in a data fusion way[20].

We believe that utilizing IR techniques on the dynamic program execution traces is beneficial to produce more relevant analysis results. Given the designed program execution scenario, dynamic program analysis builds up the connection between a software concern and its source code by quickly narrowing down the search scope to the most relevant part of the program, letting us expand the search from the
heart of the problem. IR techniques then help to facilitate this searching activity through its power and efficiency in handling and explaining large sets of data. Given enough execution trace output from dynamic program analysis, IR is able to generate a statistical model that explains the data and categorize them into various topics, where the process is referred as topic modeling, or enable us to define custom search queries to search for topic related documents. See Figure 3.5 for an overview of the implementation.

In general, IR leverages Natural Language Processing techniques (NLP)[5] to build up its document corpus. The typical steps, which are showed in Figure 3.6, consist of tokenization, stop word removal and stemming.

- **Tokenization**: Tokenization means turning a stream of characters into a stream of tokens. This is done in details steps by removing capitals, punctuation, brackets. Basically each token is a word, although the definition of a word is not straightforward especially in code corpus. A word in a coding scenario might be defined as a string of alphanumeric characters surrounded by format or syntax of the programming language used.
- **Stop Word Removal:** The purpose in stop word removal is to remove the command words like “the”, “is”, “that”, where the words does not carry any specific meaning and contribute no meaning to topic analysis. Therefore, NLTK provides a list of “stop words” which will be useful to remove all meaningless word in the corpus.

- **Stemming:** Stemming aims at identifying the basic form of each word. Words may be written in different tense and grammatical forms while still having the same meaning. For example, “saving” and “saved” can be transformed to same word “save”. Removing words in this form can help LDA achieve more accurate topic distribution.

In our case, the IR document corpus are created from program execution traces, with each document corresponding to a single class together with all its incoming methods. The generative statistical model built upon this setting enable us to perform source code topic modeling and searching on a single class level, which is helpful in the decision of class inclusion.
3.2.3.1 Latent Dirichlet Allocation

Latent Dirichlet Allocation[6] (LDA) is one of the IR techniques we chose to use for source code topic modelling. LDA is a topic model that generates topics based on word frequency from a set of documents. LDA represents documents as a mixture of topics, with each topic containing words of probability respectively. Given a set of documents, LDA is able to discover the latent topics within those documents. The general LDA analysis steps are as follows:

- Data pre-processing for analysis needs.
- Create a term dictionary, where each unique term in the documents will be assigned a unique ID.
- Generate document-term matrix serving as document corpus.
- Generate a generative statistical LDA transformation model.

As mentioned earlier, the document sets in our case come from program execution traces, which is a set of classes with each class containing all the incoming calling methods. Through LDA model, we will be able to discover the latent topics in the code subsets captured, which will be compared to see if those topics generated fit the description of our concern, or used as refinement process for finding better program entry points and execution scenarios.

3.2.3.2 Latent Semantic Indexing

Very similar to LDA, Latent Semantic Indexing[12] (LSI) is another IR techniques we chose for textual source code analysis. LSI’s general analysis process is the same as LDA, except that instead of generating a LDA model for topic modelling, LSI
generates a LSI model for customized query searching. We can translate description of a given software concern into a document. This new document then can be converted to a query and be used to search for high similarity probability documents. In our case, the search results will be a set of highly relative classes matching the description of the description of the given software concern.

3.3 Validation Strategy

Code subset verification is the last step in the methodology once we have captured a concern set based on a given software concern. During code subset minimization, we asked the question, “Do we have correctly captured all source code”. In code subset verification, we want to ask “Are all we have captured all source code for this concern?” After all, the correctness and completeness of the capture are an important criteria for the methodology.

3.3.1 Use of Independent Techniques

The easiest way of verifying the capture is through manual code inspection. The process is tedious and cumbersome, but it can serve as the most basic verification approach if there does not exist related documentation to help to decide the correctness of the capture, or if the concern being investigated is a really small concern that maps to only a few lines of code.

As we have introduced, the code minimization analysis can be divided into structural based and textual based. The IR techniques can serve as an in-process analysis as well as a different angle for the problem. The result sets returned from these two different angles can be used to validate each other independently.
3.3.2 Tool Validation

The dynamic program analysis rely heavily on Javashot for concern capturing, which relies on specific program entry points. The second validation approach is by providing another larger concern set captured under dynamic program analysis without specifying the program entry points for program execution. The concern set captured under this contains more classes and methods. We can verify the completeness of the original concern set captured by comparing these two concern sets. The java-call-graph tool can be used to generate dynamic call graphs without specifying the program entry points. The tool is introduced in section 4.2.1.

3.4 Summary and Discussion

Figure 3.7 shows a general analysis work flow for this methodology. It is an iterative translate (software concern parsing), capture (code subset analysis), and refine (topic modeling and document searching) process that will eventually help us minimize the subset of code capture to a granularity that is most relevant to the concern being specified. In summary, the methodology implementation can be described in following six steps:

1. Program Entry Point Mapping. A program entry point is the starting capturing point in the program execution trace during program execution. Given an identified software concern, the program entry points are translated, actionable keywords that best describe the concern. A software concern can map to multiple program entry points.

2. Execution Scenario Design. A program execution scenario is a test scenario designed to execute the program modules that best reflects the the given software
Figure 3.7: Analysis Work Flow

class concern, ensuring that the target concern will be captured into the program execution trace output. Using multiple execution scenarios can improve the completeness of capture at the end. We believe that, theoretically, by exhausting every possible execution scenario for program execution, the code subset relative to a certain software concern can guarantee to be captured in the execution trace output.

3. Dynamic Program Execution Trace Generation. The common approach for
dynamic program analysis is to leverage Javassist Library to enable run-time code generation. The execution trace will be generated by running the open source tool Javashot with specific program entry points.


   a) Frequency-Based Analysis. Count calling method and class frequency across multiple execution scenarios to determine the class overlap degree. Define a relation function heuristic to help decide when a class should be included into the concern set.

   b) Algorithm-Based Analysis. Implement dominator graph algorithm upon frequency analysis to include classes that have not already been identified in frequency analysis but map to the description of a given software concern.

   c) Information-Retrieval-Based Analysis. Collect and parse program execution traces down to a developer-defined granularity such as a class level granularity to form a set of documents for analysis. Leverage information retrieval techniques such as LDA and LSI for source code topic modeling and customized query searching.

5. Concern Refinement. The analysis output return by IR analysis will be used for concern refinement for next more accurate dynamic program analysis.

6. Verification of Capture. Verify the concern set captured by either through manual code inspections when the concern being investigating is small or use dynamic call graph verification approach for a more complex concern.

There are similar approaches as introduced in literature review section by combining static, or dynamic program analysis with IR techniques for source code local-
ization. In summary, there are three major differences between this approach and previous works done:

- The dynamic program analysis in this approach focuses on producing only relevant execution traces with well-designed scenarios, while other approaches compare relevant execution traces captured with some irrelevant program outputs to determine relevant code subsets.

- Besides test execution scenarios, program execution is guided through translated program entry points from an given software concern to enable the analysis starts from the heart of the problem.

- Instead of analyzing the entire software program like other approaches do, or combining source code with some other forms of documentation, the IR techniques proposed in this methodology only analyze the relevant execution trace outputs generated by the dynamic program analysis. It can significantly reduce the noise that produced by the irrelevant information, making the analysis model clean and focused on the target concern. In addition, the document format designed as each document is a class containing all in-coming methods in this methodology make it efficient for software concern mapping.
Chapter 4

Case Study Results

This section introduces the results after applying the methodology developed in Chapter 3 on some real-world open source projects for software concern capturing. Each specific task that need to be performed to conduct a concern capturing has been incorporated into a Python GUI tool named CCAP (concern capture). The source code of the tool is currently hosted on GitHub repository named concern capture (https://www.github.com/ctfu/concernCapture).

4.1 Systems Under Study

We have conducted case studies on multiple open-source projects. This section shows the analysis only of the major two projects, one is JHotDraw, the other one is ArgoUML. The analysis on JHotDraw was on earlier stage, the techniques used are mostly about finding suitable tools and developing appropriate code for constructing customized dynamic call graphs under dynamic program analysis. The analysis on ArgoUML was more thorough, which we have applied all techniques mentioned in the methodology for this case study.
4.1.1 JHotDraw

JHotDraw is an open-source Java GUI framework for developing technical and structured graphics. The project was developed under the exercise of some well-known design patterns. It then become popular due to its powerful features. The code was originally developed by Erich Gamma and Thomas Eggenschwiler. Then the project was open source for developers to freely contribute. To see more details on JHotDraw, see the link http://www.jhotdraw.org/ to JHotDraw official site.

4.1.2 ArgoUML

ArgoUML is a powerful UML modelling tool developed in Java and released under open-source Eclipse Public License. The project was originally developed at UC Irvine by Jason E. Robbins, now the project is hosted on http://www.tigris.org/.

4.2 Analysis Results

The analysis result for JHotDraw and ArgoUML is presented in this section. The tools we used for constructing dynamic call graphs will also be introduced here. In addition to these tools, we have also developed some Python scripts to assist with the analysis process.

- **tracer.py**: a program to process and customize the dynamic call traces generated by Javashot and builds up an visual call graph.

- **tracerFreq.py**: Similar to trace.py, it also adds in the frequency output while producing the graph.

- **frequency.py**: a program to assist in the frequency analysis.
• **ir.py**: a program that implemented with LDA and LSI functionality for IR-base analysis.

To see detailed usage of these programs, please refer to my GitHub repository https://github.com/ctfu/concernCapture.

### 4.2.1 Dynamic Call Graph Tools

Based on the methodology developed, we investigated two dynamic call graph generation tools, Javashot and Java-call-graph. While one captures the call traces with specifying an program entry point, the other one captures the call traces targeting the entire software program. The reasons for this, as we have mentioned in details in the methodology section, is to have a way to verify the completeness of our final capture.

#### 4.2.1.1 Javashot

Javashot, known as Java Dynamic Call Graph, leverages Java instrumentation capabilities to capture the dynamic execution flow of the program. Javashot’s runtime code instrumentation is done through Jboss-javassist (Java Programming Assistant), a Java bytecode engineering tookit that makes bytecode manipulation simple over JVM. It offers two levels of API, bytecode level and source level, for editing bytecodes in Java. To see more details of this toolkit, please refer to [http://jboss-javassist.github.io/javassist/](http://jboss-javassist.github.io/javassist/). Javashot generates threads of .dot files during the execution of the program, which can be visualized using Graphviz ([http://www.graphviz.org/](http://www.graphviz.org/)).

Some configuration will need to be set up in order to successfully run the program. Specifically, it requires you to set up a specific program entry point in the
javashot.properties file prior to executing the program. To see a detailed configuration setting, please refer to https://code.google.com/archive/p/javashot/. The newest Javashot source code can be downloaded on Github https://github.com/arebya/javashot.

4.2.1.2 Java-call-graph

Similar to Javashot, Java-call-graph is a suite of programs for generating static as well as dynamic call graphs for Java developers. It also relies on Javassist for runtime code instrumentation. The major difference of this tool is that it captures the dynamic execution flow of the program starting with the main entry points, which potentially could capture the entire program. However, as of writing this thesis, this tool still has problems in handling multi-threading programs and exceptions in their dynamic call graph generator.

4.2.2 JHotDraw Analysis

Inspired by Robillard in his paper [22] investigating Undo feature in JHotDraw, we decided to also do a case study on capturing the undo concern, so that we will have some way of verifying our concern set captured at the end.

4.2.2.1 Program Entry Points

The undo functionality is implemented as one of the many commands in JHotDraw involving several interfaces, abstract classes and concrete classes. With little help of manual code investigation, we decided to select UndoCommand as our program entry point.
4.2.2.2 Program Execution Scenarios

We executed JHotDraw over Javashot under seven different execution scenarios about drawing a graphic involving simple to complex drawing activities. One of the drawing scenario containing following simple steps:

1. Create a new file
2. Draw a simple rectangle
3. Fill the rectangle with red color
4. Undo last operation
5. Exit the program

4.2.2.3 Frequency Analysis Result

Figure 4.1 shows the dynamic call graph rendered for the above mentioned execution scenario by processing the execution traces through tracer.py. The nodes are classes that executed during dynamic program execution. Edge shows the method calls, where we have combined multiple method calls by indication of edge thickness. Edge label shows the first method that have been called between two classes. By observing classes that appear frequently across multiple execution scenarios, we set a cutting point over class StandardDrawingView (node colored red) based on our heuristic studies on the source code to reduce our capture to a minimal level. By comparing the final concern set captured in this dynamic call graph with Canfora and Cerulo’s [7] undo concern study (see Figure 4.2), we can confirm that we have captured all the necessary classes involved in the undo concern.
Figure 4.1: JHotDraw Undo Concern Call Graph

Figure 4.2: JHotDraw Undo Concern Participants in Canfora and Cerulo [7]
4.2.2.4 Algorithm Analysis Result

The dominator tree constructed for the above mentioned execution scenario is a flat tree with three levels (see Figure 4.3), where most of the nodes dominate themselves. We couldn’t identify extra nodes in this scenario, which help to confirm the correctness of our previous capture, where most of the related classes have already included in the result set.

Figure 4.3: Dominator Tree for JHotDraw Undo Based on Frequency Analysis. The colored nodes represent classes present in all execution traces collected.
4.2.2.5 Information Retrieval Analysis Result

Since the structural way of analysis yielded a promising capture, we are interested to compare the result with textual analysis. Table 4.1 shows the LDA analysis output of discovering 10 topics with each topic containing three words. The topics return give us a general idea of the capture, which is about drawing and displaying a figure, undo or redo the figure. And topic 2 gives the best matches. These indicate that we have the right capture. However, each topic word probability is relatively low. The reason is either because the input data is low, or the calibration of the model parameter is off.

<table>
<thead>
<tr>
<th>Topic ID</th>
<th>Word</th>
<th>Word</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.127*“figure”</td>
<td>0.064*“view”</td>
<td>0.064*“with”</td>
</tr>
<tr>
<td>1</td>
<td>0.107*“figure”</td>
<td>0.094*“undo”</td>
<td>0.048*“redo”</td>
</tr>
<tr>
<td>2</td>
<td>0.068*“command”</td>
<td>0.068*“with”</td>
<td>0.068*“executable”</td>
</tr>
<tr>
<td>3</td>
<td>0.078*“executable”</td>
<td>0.078*“command”</td>
<td>0.078*“with”</td>
</tr>
<tr>
<td>4</td>
<td>0.101*“command”</td>
<td>0.081*“view”</td>
<td>0.074*“selection”</td>
</tr>
<tr>
<td>5</td>
<td>0.111*“attribute”</td>
<td>0.082*“command”</td>
<td>0.067*“figure”</td>
</tr>
<tr>
<td>6</td>
<td>0.113*“event”</td>
<td>0.097*“drawing”</td>
<td>0.096*“change”</td>
</tr>
<tr>
<td>7</td>
<td>0.075*“view”</td>
<td>0.059*“display”</td>
<td>0.059*“box”</td>
</tr>
<tr>
<td>8</td>
<td>0.119*“view”</td>
<td>0.076*“command”</td>
<td>0.070*“executable”</td>
</tr>
<tr>
<td>9</td>
<td>0.114*“executable”</td>
<td>0.114*“view”</td>
<td>0.113*“with”</td>
</tr>
</tbody>
</table>

Table 4.1: LDA Analysis Output For JHotDraw. The number next to each word indicates how likely each word is related to its topic.

The LSI analysis returned a similar capture as the frequency-based analysis, using the query “undo”. (See Table 4.2) It shows additional classes for this execution such as AttributeFigure, DecoratorFigure that did not appear in the dynamic call graph. They are the classes being truncated by the cutting point Standard-DrawingView. The interesting result of this table is that there are also negative probability classes, where a negative probability indicates the irrelevance. Standard-
**DrawingView** almost has the highest irrelevant probability of -0.00247855, which is a good indications that we have identified the right cutting point.

<table>
<thead>
<tr>
<th>Class/Method</th>
<th>Activity</th>
<th>Activity Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>PasteCommand_UndoActivity</td>
<td>34</td>
<td>0.893948</td>
</tr>
<tr>
<td>UndoManager</td>
<td>4</td>
<td>0.87829</td>
</tr>
<tr>
<td>ChangeAttributeCommand_UndoActivity</td>
<td>6</td>
<td>0.735755</td>
</tr>
<tr>
<td>UndoCommand</td>
<td>0</td>
<td>0.562284</td>
</tr>
<tr>
<td>UndoableAdapter</td>
<td>5</td>
<td>0.448267</td>
</tr>
<tr>
<td>DrawApplication</td>
<td>2</td>
<td>0.335613</td>
</tr>
<tr>
<td>AttributeFigure</td>
<td>9</td>
<td>0.250315</td>
</tr>
<tr>
<td>RedoCommand</td>
<td>31</td>
<td>0.148865</td>
</tr>
<tr>
<td>ChangeAttributeCommand</td>
<td>33</td>
<td>0.147206</td>
</tr>
<tr>
<td>UndoableCommand</td>
<td>18</td>
<td>0.141681</td>
</tr>
<tr>
<td>CommandMenu</td>
<td>17</td>
<td>0.133428</td>
</tr>
<tr>
<td>PasteCommand</td>
<td>23</td>
<td>0.0320243</td>
</tr>
<tr>
<td>DecoratorFigure</td>
<td>8</td>
<td>0.025318</td>
</tr>
<tr>
<td>AbstractCommand_EventDispatcher</td>
<td>3</td>
<td>0.0214342</td>
</tr>
<tr>
<td>AbstractFigure</td>
<td>11</td>
<td>0.0180849</td>
</tr>
<tr>
<td>AnimationDecorator</td>
<td>35</td>
<td>-0.00180239</td>
</tr>
<tr>
<td>StandardDrawingView</td>
<td>16</td>
<td>-0.00247855</td>
</tr>
<tr>
<td>StandardDrawing</td>
<td>14</td>
<td>-0.00640339</td>
</tr>
<tr>
<td>DrawingChangeEvent</td>
<td>15</td>
<td>-0.00845594</td>
</tr>
<tr>
<td>SendToBackCommand</td>
<td>29</td>
<td>-0.00888022</td>
</tr>
</tbody>
</table>

Table 4.2: LSI Partial Analysis Output For JHotDraw

### 4.2.3 ArgoUML Analysis

As we know the “save” feature is a very important feature in all most every tool. Depends on the specific application, the code subset involved of its saving mechanism could be different. In general, it involves with a representation of a file system and the actual saving functionality. Since we have already have some knowledge of the how saving works, we are interested to capture the **project saving concern** in ArgoUML.
4.2.3.1 Program Entry Points

Given a project saving concern, we parse down the concern context into some actionable keywords such as “action save” and “persist”, that are search-able in customized filter in static program analysis. We found that there are multiple classes that support the saving functionality. To simplify the problem, here we only identify one program entry point, which is \textbf{ActionSaveProject} class.

4.2.3.2 Program Execution Scenarios

ArgoUML is powerful in creating different types of UML diagrams. To fully capture the context of how a project is saved, we have designed four execution scenarios to mimic some simple operations of a vending machine, which are

- Use Case Diagram: a diagram that models the interaction between a user and the elements of a system. See Figure 4.4.

- Class Diagram: a static structure representation the system modeled using objects of Classes and their interactions. See Figure 4.5.

- State Diagram: a type of diagram that models the behaviors of a system. See Figure 4.6.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{use_case_diagram.png}
\caption{Use Case Diagram Scenario}
\end{figure}
• Sequence Diagram: an interaction diagram that shows how objects of a software system interacts with each other arranged in time sequence. See Figure 4.7.
4.2.3.3 Frequency Analysis Result

With a good program entry point, some well-designed execution scenarios and proper configuration of Javashot, we generated a total of four unique dynamic call traces, each set of call trace corresponds to one execution scenario. After some pre-processing of the call traces generated such as file concatenation and file conversion, the data set is ready for frequency analysis by applying a Python script to count the unique frequency for each class executed across all execution scenarios. Table 4.3 shows a sample frequency analysis output.

4.2.3.4 Algorithm Analysis Result

By identifying those high frequency classes in the frequency analysis output i.e. **ZipFilePersister**, and **ZargoFilePersister**, we can further apply dominator algorithms on those classes to bring in all their dominators. However, in this case scenario, as Figure 4.8 indicates, their dominators are already included in the captured. In gen-
### Table 4.3: Partial Frequency Analysis Output For State Diagram Scenario

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZipFilePersister</td>
<td>4</td>
</tr>
<tr>
<td>ZargoFilePersister</td>
<td>4</td>
</tr>
<tr>
<td>XmlFilePersister</td>
<td>4</td>
</tr>
<tr>
<td>XmiFilePersister</td>
<td>4</td>
</tr>
<tr>
<td>UpArrowIcon</td>
<td>4</td>
</tr>
<tr>
<td>UmlFilePersister</td>
<td>4</td>
</tr>
<tr>
<td>UMLTreeCellRenderer</td>
<td>4</td>
</tr>
<tr>
<td>TypeThenNameOrder</td>
<td>4</td>
</tr>
<tr>
<td>SwingWorker_2</td>
<td>4</td>
</tr>
<tr>
<td>SwingWorker_1</td>
<td>4</td>
</tr>
<tr>
<td>ProjectFileView</td>
<td>4</td>
</tr>
<tr>
<td>ProjectBrowser_TitleHandler_1</td>
<td>4</td>
</tr>
<tr>
<td>UMLToDoItem</td>
<td>3</td>
</tr>
<tr>
<td>UMLListCellRenderer2</td>
<td>3</td>
</tr>
<tr>
<td>UMLLinkedListCellRenderer</td>
<td>3</td>
</tr>
<tr>
<td>SelectionNodeClarifiers2</td>
<td>3</td>
</tr>
<tr>
<td>ResourceLoader</td>
<td>3</td>
</tr>
<tr>
<td>ListSet</td>
<td>3</td>
</tr>
<tr>
<td>FigNodeModelElement</td>
<td>3</td>
</tr>
<tr>
<td>KindsMDRImpl</td>
<td>2</td>
</tr>
<tr>
<td>FigEdgeModelElement</td>
<td>2</td>
</tr>
<tr>
<td>FigAssociation_EndDecoration</td>
<td>2</td>
</tr>
<tr>
<td>FigAssociationEndAnnotation</td>
<td>2</td>
</tr>
<tr>
<td>FigAssociation</td>
<td>2</td>
</tr>
<tr>
<td>DiagramSettings</td>
<td>2</td>
</tr>
<tr>
<td>ModelManagementHelperMDRImpl</td>
<td>1</td>
</tr>
<tr>
<td>FigTransition</td>
<td>1</td>
</tr>
<tr>
<td>FigGeneralization</td>
<td>1</td>
</tr>
<tr>
<td>FigActor</td>
<td>1</td>
</tr>
<tr>
<td>CompartmentFigText</td>
<td>1</td>
</tr>
</tbody>
</table>

General, with dominator analysis, our capture will be more complete and less reliant on element name matching.
Figure 4.8: Dominator Tree for ArgoUML Based on Frequency Analysis
4.2.3.5 Information Retrieval Analysis Result

Before we can apply LDA and LSI to the program execution traces collected, we need to filter our data sets and decide what data should be put into one document so that the analysis will best fit into our needs for concern capturing. Consider our goal of capturing is to return a most related set of classes to the given concern, we define each class and all its in-coming methods as one document unit. With each execution scenario, we then can form a set of documents containing data that are directly related to the given concern. Depending on specific analysis needs, if a larger set of data is needed, more execution scenarios can be added for the program to execute, generating more execution traces under each scenario specified. From our experience, the more related data fed to LDA or LSI model, the more reliable the result.

The section here shows a simple example performing the analysis on the scenario of drawing a state diagram as Figure 4.6. There are 365 execution trace files in these scenarios. Each execution trace file is a dot file generated by Javashot containing some method calling traces. We first need to concatenate all these files and pre-process it to the format that fit for the analysis. We then apply the typical LDA/LSI analysis steps to perform the analysis. For LDA, we conduct the analysis for discovering 10 topics out of the data set and configure every topic to contain 3 words. Table 4.4 shows a sample output.

From the topic output, we can immediately get some insights to our save project concern capture, e.g., topic 0 indicates the saving project involves swing worker component; topics 1 and 2 talk about element naming and tree structure, which also make sense because saving a project requires us to first specify a name for the project, then save the project to some local directory. In ArgoUML, when browsing the file system
Table 4.4: LDA Analysis Output For State Diagram Scenario. The number next to each word indicates how likely each word is related to its topic.

to save a file, a certain directory is represented as a tree structure. Topic 7 gives the most relevant information, it says that to save a file, we need to make use of a persister in the system. It is consistent with our frequency analysis output shown in Table 4.3, that most of the high frequency classes we captured across multiple execution scenarios has something to do with file persister.

Through LSI, we can create a customized document to query the LSI model. Table 4.5 shows a partial list of topmost related documents (classes) with probabilities respectively that are related to the query defined as **save project**.

It is obvious that the most related document is **ActionSaveProject**, it also serves as our program entry point for the analysis. Other documents related for saving project are **ProjectBrowser**, **ProjectFileView**, for browsing through file system to select a destination, and **SaveSwingWorker**, **CmiFilePersister**, **ZipFilePersister**, **UmlFilePersister** when actually saving the file.
Table 4.5: LSI Partial Analysis Output For State Diagram Scenario

4.3 Discussion and Limitations

As the analysis on JHotDraw and ArgoUML indicates, the methodology shows a promising result for capturing a given software concern, even when the test data
set is relatively small. Each case study was performed on limited number of program entry points and execution scenarios. JHotDraw was analysis on single program entry point and seven execution scenarios. ArgoUML was analysis on single program entry point and four execution scenarios. We believe that the accuracy of a given concern capture under this approach could be significantly improved if working on larger data sets by executing the program on more program entry points and more execution scenarios.

The premise for a reliable analysis is to have proper program entry points and good execution scenarios. These help to narrow down the scope of a given concern and guide the analysis to start from the heart of the problem. Thus, we make a general assumption that, with the proper tool support, the users are capable of identifying concern related program entry points and designing functional execution scenarios.

Dynamic program analysis and dynamic call graph construction is the foundation of all later code subset minimization techniques. The dynamic program analysis itself heavily relies on Javashot, a tool which we found has potential threading problems when handling multi-threading application. The threading problem appears also to be the problem for the validation tool– Java-call-graph we have experimented on, which limits our code captured validation done in this analysis, through comparing the capturing result between IR based textual analysis and the rest of the structural base analysis.

The frequency-base analysis relies on identifier matching for making a better decision whether a class should be included in the result set. This could easily introduce false positive capture regardless of not having a good naming convention for the program. The common challenge in IR based analysis is to set the proper parameters such as number of topics and number of topic words to generate an analysis model that best represents the problem. The analysis output could have huge difference
depending on specific analysis purpose and how data sets are defined. And often
times, it is a long trial and error process to get the most desired output.
Chapter 5

Conclusion and Future Works

5.1 Summary

Software concern capturing has always been an important yet challenging topic in software engineering. Being able to trace a semantic concern to its actual code offers significant benefits in reducing software maintenance costs, improving code reusability, and increasing overall life span of a functional software. To achieve code localization, researchers has investigated amount of efforts in various analysis techniques typically involving static program analysis, dynamic program analysis, IR techniques and hybrid techniques among these three.

The research methodology proposed in this thesis utilize tool-assisted dynamic program analysis as our analysis foundation. We introduce the concept of program entry point and program execution scenarios to guide the dynamic program execution to allow us quickly get to the relative portion of the program. We also introduce frequency-based, algorithm based and IR based code minimization analysis to help us achieve the level of capture to the granularity most related to the given software concern. The validation strategy for concern set capture is done both by comparing
the analysis result between structural and textual execution trace analysis, and tool validation for dynamic call graph construction.

We have applied the methodology developed to two real world open source projects, JHotDraw and ArgoUML. The methodology shows a promising concern capturing result even when the input data set is relatively small. We believe that the accuracy of a given concern capture under this approach can be significantly improved if working on larger data sets by including more program entry points and execution scenarios.

There are three major differences between this approach and the similar previous works introduced in the literature review section. First, our dynamic program analysis focus on producing only relevant execution traces without the need to compare the result with irrelevant capture under pre-designed unrelated execution scenarios. Second, we introduce the concept of program entry points to guide the dynamic program execution to force our analysis starting within the heart of the problem. Third, unlike approaches that utilize IR techniques as an independent analysis, IR based analysis works as one of the in-process steps in our concern capturing. In addition to this, we have a special design format for documents that will become the source of the IR analysis. We focus on only the relevant execution traces produced by dynamic program analysis rather than combining other source of documents such as requirements documents and inline code comments as IR’s source input. It significantly reduces the noise that will be produced by the less irrelevant information. In this way, the IR analysis model generated is clean and more focused on the target concern being captured.
5.2 Future Work

The analysis such as frequency based analysis relies on identifier matching techniques, which could easily produce false positive code identification. In addition to applying algorithm analysis, IR techniques can also be used to reduce the false positive rate on identifier matching. The analysis result could be further improved by behavior matching. An example is performing static analysis on the classes among identified concern set to determine methods that have modified an object’s state, where that object belongs to a given software concern. Those classes that involve in changing the object’s state could indicate computations contributing to that concern.

Since a source for the input for our IR analysis are classes containing all their executed methods, we are able to efficiently search for top related classes that are most related to a customized document query. In the future, analysis based on these relationships can be used in document clustering analysis for further concern grouping and slicing. Once a concert set is captured, program migrating techniques could be developed to better re-used those classes for systems that intend to have the similar functionality.


Bibliography


