SOFTWARE ANALYTICS FOR IMPROVING PROGRAM COMPREHENSION

A Dissertation
IN
Computer Science

Presented to the Faculty of the University
of Missouri–Kansas City in partial fulfillment of
the requirements for the degree

DOCTOR OF PHILOSOPHY

by
RAKAN ALANAZI

B. S., Northern Border University, Rafha, Saudi Arabia, 2012
M. S., University of Missouri–Kansas City, MO, USA, 2016

Kansas City, Missouri
2021

SOFTWARE ANALYTICS FOR IMPROVING PROGRAM COMPREHENSION

Rakan Alanazi, Candidate for the Doctor of Philosophy Degree

University of Missouri–Kansas City, 2021

ABSTRACT

Program comprehension is an essential part of software development and maintenance. Traditional methods of program comprehension, such as reviewing the codebase and documentation, are still challenging for understanding the software's overall structure and implementation. In recent years, software static analysis studies have emerged to facilitate program comprehensions, such as call graphs, which represent the system's structure and its implementation as a directed graph. Furthermore, some studies focused on semantic enrichment of the software system problems using systematic learning analytics, including machine learning and NLP. While call graphs can enhance the program comprehension process, they still face three main challenges: (1) complex call graphs can become very difficult to understand making call graphs much harder to visualize and interpret by a developer and thus increases the overhead in program comprehension; (2) they are often limited to a single level of granularity, such as function calls; and (3) there is a lack of the interpretation semantics about the graphs.

In this dissertation, we propose a novel framework, called CodEx, to facilitate and accelerate program comprehension. CodEx enables top-down and bottom-up analysis of the system's call graph and its execution paths for an enhanced program comprehension experience. Specifically, the proposed framework is designed to cope with the following techniques: multi-level graph abstraction using a coarsening technique, hierarchical clustering to represent the call graph into subgraphs (i.e., multi-levels of granularity), and interactive visual exploration of the graphs at different levels of abstraction. Moreover, we are also worked on building semantics of software systems using NLP and machine learning, including topic modeling, to interpret the meaning of the abstraction levels of the call graph.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of graduate studies, have examined a dissertation titled "Software Analytics for Improving Program Comprehension," presented by Rakan Alanazi, candidate for the Doctor of Philosophy degree, and hereby certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Yugyung Lee, Ph.D., Committee Chair
Department of Computer Science & Electrical Engineering

Sejun Song, Ph.D.
Department of Computer Science & Electrical Engineering

Raveen Rao, Ph.D.
Department of Computer Science & Electrical Engineering

Baek-Young Choi, Ph.D.
Department of Computer Science & Electrical Engineering

Zhu Li, Ph.D.
Department of Computer Science & Electrical Engineering

CONTENTS

ILLUSTRATIONS

TABLES

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Dr. Yugyung Lee. This dissertation would not have been possible without your guidance and persistent help. I appreciate all of your time and effort in assisting me with my research. I've learned a lot from you and gained a lot of valuable experience. I can confidently say I would not be where I am currently as a scholar if it were not for you. You have also made it a priority to teach me to do more than just research. You help me get my name into publications, so I may stand on my own in the future.

I also want to thank the members of my committee Dr. Sejun Song, Dr. Praveen Rao, Dr. Baek-Young Choi, and Dr. Zhu Li, for giving their time to advise, listen, and support. I also would like to thank all the UMKC Distributed Intelligent Computing Association (UDICA) members for their kind assistance during my study. Special thanks go to my colleagues Gharib Gharibi and Vijay Walunj for their support and discussions.

Finally, my sincere thanks go to my family. I cannot express my thanks to my parents for their love, prayers, and continuing support over my life. My deepest thanks and gratitude to my wife and son, Anas for being in my life and part of this journey.

CHAPTER 1

INTRODUCTION

1.1 Program Comprehension

Comprehending the implementation and structure of a system is required before applying any proper modification or enhancement to the system. Thus, program comprehension is an essential and costly activity in software maintenance [118]. Studies report that program comprehension is an intensive and time-consuming process. Around 60% of the software engineer's time is spent in this task [27,122,140]. The developer usually uses different artifacts to help him understand the system. These artifacts can be divided into two types: code artifacts and documentation artifacts. Developers often read the system's documentation, (i.e., a high-level artifact). However, software documents are often outdated, i.e., current documentation does not match the current software implementation. Thus, code artifact is considered as the only trusted source to understand the software. Therefore, in this dissertation, we focus mainly on the source code as well as on these entities that can be extracted from the source code and helpful in understanding the source code.

Program comprehension approaches can be divided into different categories [36]. We describe the most important categories top-down approach [9, 102], bottom-up approach [120] and a combination of the two [124].

In the top-down comprehension approach, the software engineer gains an understanding of the source code by relying on his prior domain knowledge and experience. The developer starts with the most abstract problem domain concepts and builds a set of assumptions or hypotheses. Then, he verifies the validity of his hypotheses by inspecting the source code and other artifacts. In the end, a hierarchy of assumptions is built until the low-level hierarchy is matched with the source code.

In the bottom-up comprehension approach, the developer is not familiar with the problem domain. He starts by reading the source code line by line, understanding the behavior of small pieces of code, and then grouping these small chunks of source code to build higher levels of abstraction. This process is repeated until the entire program is understood [75, 120].

The integrated approach, which involves the top-down, bottom-up methods, is commonly used when the developer tries to understand a large-scale system. The reason behind this is that some parts of the source code may be familiar to the developer because of previous experiences while other parts of the code may be completely new [132].

1.2 Software Analytics

During software development and maintenance activities, a significant amount of data can be collected including requirements, source codes, bug reports, test cases, execution traces/logs, etc. However, This data is hard to manage and understand by humans. Thus, researchers in the field of software engineering (SE) have turned their attention to studies based on data mining (DM) and machine learning (ML). Applying data mining

enables the discovery of useful knowledge and hidden patterns from collected SE data. There are many research topics in software engineering and data mining that can provide more insight and support decision-making in those areas. As shown in Figure 1, the intersection between data mining, software engineering, and statistics/math.

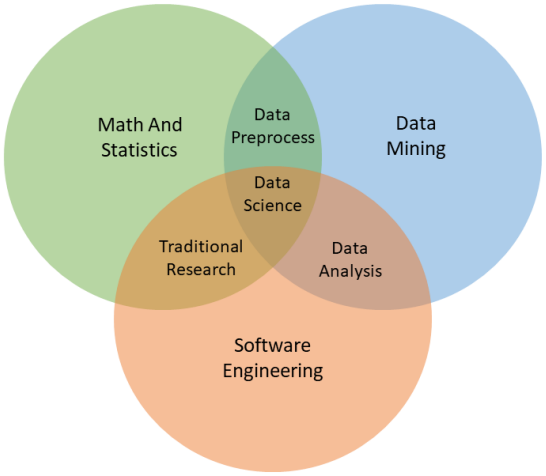


Figure 1: Intersection of DM and SE with Other Areas of the Field

The main difference between software analytic and direct software analysis is that software analytics performs further advanced steps rather than just providing straightforward insight. As explained by Hassan [54], in order to facilitate decision making, software analytics must provide visualization and useful interpretation of insights.

1.3 Semantic and Structural Analysis

Since source code is the most reliable and important input for program comprehension, several methods and techniques used the source code as the only trusted source

to understand the software. These techniques can be divided into two different categories: syntactic and semantic. The syntactic approaches are based on static program structural information (e.g., classes, methods, and attributes) and its dependencies (e.g., inheritance, method calls, references). A number of syntactic-based techniques have been proposed [60, 90]. The semantic approaches, on the other hand, consider analyzing and extracting domain knowledge of the system by utilizing the source code comments and identifier names [64, 70, 116, 125]. In this dissertation, we focus on investigating the combination of using semantic and structural information of programs to support the comprehension tasks.

1.4 Static Analysis

In this dissertation, we choose to focus on static analysis to comprehend the software system and identify the functionality of the system. The main reason for this is that by static analysis, we can preserve the overall structure of the system and model it into different representations such as a call graph. Moreover, we can examine all possible execution paths, which is not the case in dynamic analysis. Dynamic analysis can only provide a partial picture of the system, i.e., the results obtained are valid for the scenarios that were exercised during the analysis. Therefore, obtaining all possible execution paths of the system is very difficult. Especially, execution paths associated with conditional statements (if-then-else) may require the developer to provides a set of different inputs. That is why dynamic analysis suffers from incomplete execution paths.

1.5 Contributions

The main intent of this dissertation is to support program comprehension using visual analysis and exploration of a software system. We first show how a system's call graph can be used as a starting point for program comprehension by developing an interactive visualization tool that can analyze the source code of a system and visualize its structure in multi-level of abstractions. Second, We develop a data-driven approach of static analyzes to mine a system's execution paths and determine the implementation detail of a system. Many of the execution paths perform similar functionalities and share a huge number of functions with other paths. Therefore, to manage or handle this large volume of execution paths, we apply clustering techniques to group these paths based on similar functionalities they share. Third, To further support the comprehension process, an interactive clustering-based visualization tool was proposed to facilitate the process of analysis and exploration of clustered paths. It provides a mapping technique to link the resulted clusters to the overall system structure using the call graph representation. Finally, To interpret the meaning of the resulted clusters, we apply topic modeling techniques to understand cluster's functionality by generating labels that reflect the meaning of each cluster.

Our overarching goal is to assist software developers in understanding the software system from a high level of abstraction to a low level of implementation with the ability to focus on particular parts of the system individually. To validate our approach and tool support, we built a framework that implements the execution path mining and the visualization aspects. It can analyze the codebase of a system and construct the static call

graph for a system, cluster the execution paths of the call graph into hierarchical abstractions, and label the clusters according to their major functionality. The framework also lets the users visualize the structure of the system in multi-level abstractions. Moreover, it provides different features including filtering, search, and quantitative information to aid the comprehension process. The framework would assist the developers/maintainers in the following ways:

- Allow the developers to understand the implementation and the structure of a software system using the call graph at multi-level of abstractions.
- Allow the developers to bridge the cognitive gap between the system's overall functionality and its implementation by automatically mapping high-level system functionality to its low-level implementation using our proposed clustering techniques.
- Allow the developers to visually analyze and interpret the clustered execution paths by labeling and linking the clustering results to the overall system structure.

Moreover, we conducted a user study of 18 software engineers from more than 11 industries who carried out several tasks using our system and then answered a survey. The results demonstrate that our approach is feasible to automatically construct multi-level abstractions of the call graph and hierarchically cluster them into meaningful abstraction. Figure 2 depicts the high level of the approach workflow. To summarize, the main contributions of this dissertation include:

- Developer an automated language independent approach for analyzing the source

code of a system and visualizing its structure in multi-level abstractions using the call graph [2].

- Develop an automated data-driven approach to bridge the cognitive gap between the high-level functionality of a software system and its low-level implementation [2].
- To interpret the meaning of the resulted clusters, we apply topic modeling techniques to understand clusters functionality by generating labels that reflect the meaning of each cluster.
- Develop an interactive clustering-based visualization tool to facilitate the process of analysis, exploration and mapping of resulted clusters to the overall system structure [2].

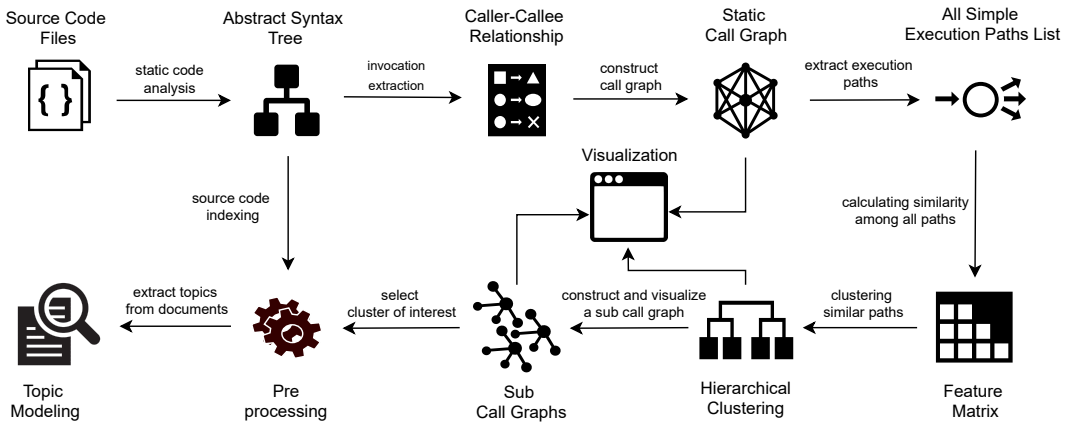


Figure 2: High-level of the Approach Workflow.

1.6 Thesis Organization

The remainder of this dissertation is organized as follows. **Chapter 2** describes our attempts in understanding software through visualization. We show how a system's call graph can be used as a starting point for program comprehension. We propose an interactive visualization tool that can analyze the source code of a system and visualize its structure in multi-level abstractions using the call graph. **Chapter 3** of this dissertation deals with a program comprehension solution that is based on static trace analysis. We introduce the clustering technique using machine learning to group the execution paths of a system based on similar functionalities they share to support the comprehension process. **Chapter 4** addresses the need for path reduction techniques in program comprehension. We discuss the limitations of existing solutions and introduce a new reduction technique that enables analyzing the execution paths of a medium or large-scale software system. **Chapter 5** describes our approach of analyzing and interpreting the meaning of the clustered call graphs using topic modeling technique along with our visual abstraction technique of the call graph. **Chapter 6** describes the implementation of a visual clustering-based exploration tool that supports our scalable clustering approach. In other words, We implemented CodEx, a scalable tool that supports all the aforementioned applications. We also conducted a user study to assess the usefulness and usability of the tool.

CHAPTER 2

MULTI-LEVEL CALL GRAPH FOR PROGRAM COMPREHENSION

2.1 Introduction

Program comprehension is an imperative prerequisite for software reuse, debugging, testing, maintenance, and evolution [29]. In order to facilitate the task of understanding the software and its implementation, developers often read the system's documentation, i.e., a high-level description of the software system, and then manually map their understanding of the system to its low-level implementation. However, software documents are often outdated, i.e., current documentation does not match the current software implementation [91]. As a software system evolves and increases in size and complexity, understanding its implementation and structure becomes an even more challenging and time-consuming task. Manually mapping the high-level functionality to its low-level implementation is expensive, time-consuming, and error-prone. Therefore, software developers use analysis tools to understand and gain more knowledge about the system's implementation. One of the techniques used by software developers to understand the functionality and structure of a system is call graphs. A call graph depicts the system structure in terms of function calls or operational invocations [42, 57, 95]. Call graphs [94, 112] aid to facilitate software comprehension and operational analysis tasks and can enhance software-related activities, such as debugging and maintenance [3]. In the literature [42, 59, 62, 73, 119], various techniques have been proposed to visualize the

call graph of a system. These approaches are promising but limited in several aspects:

- Visualize a portion of the call graph [59, 73]. Thus can not depict the overall structure of a system.
- Limited to a single level of granularity (i.e., function level) [42, 119] and thus may not well support different maintenance tasks requiring understanding either fine or coarse grain, or both level.
- Generate the call graph in static format [59, 62], which makes the exploring and understand process difficult.

To overcome these issues, we develop a call graph visualization tool with a dynamic, browser-based user-friendly interface. The tool provides a set of features to further support the comprehension process, keep the amount of information in the graph manageable, and reduce the effort to understand it. Moreover, it provides different levels of abstraction to aid the developers in understanding the implementation and the structure of a software system using the call graph.

2.2 Background

In this section, we present a brief background on the topic of call graphs and define some of the related terms repeatedly used in the rest of the chapters.

Call Graph has been widely used to facilitate understanding the structure and execution flow of software systems [14, 47, 112]. A call graph can be dynamic [45], constructed at runtime, or static [94], constructed at compile time. The static call graph

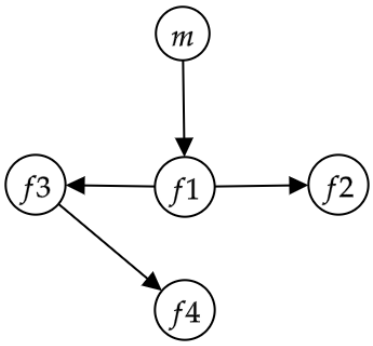
```java
public class HelloWorld{

    public static void main(String []args){
       System.out.println("Hello");
       f1();
    }

    public static void f1(){
       System.out.println("Rakan");
       f2();
       f3();
    }

     public static void f2(){
       System.out.println("!");
    }
     public static void f3(){
       f4();
    }
      public static void f4(){
       System.out.println("!!");
    }
}
```

(a) Source Code



(b) Call Graph

Figure 3: Representing Source Code to Call Graph

can be a very useful technique in reverse engineering and software maintenance [77]. A significant advantage of the static call graph over the dynamic call graph is that it can be constructed without the need of executing the program, and all the dependency information is gathered directly from the source code of the system, which is not the case in the dynamic call graph. The dynamic call graph would require the developer to know what inputs data to provide, and how to execute the system. Moreover, obtaining all possible execution paths of the system is very hard. Especially, execution paths associated with conditional statements (if-then-else) may require the developer to provides a set of different inputs. That is why dynamic call graph suffers from incomplete executions path [9, 37, 67]. In contrast, the static call graph represents all possible execution paths of the system. Our research focuses on static call graphs, and hereafter we use the term 'call graph' to refer to 'static call graph' for simplicity unless otherwise distinguished.

In a software system, a static call graph [94] is defined as a directed graph $\vec{G}$ = (V, E) where V is the set of v entities, and E is the set of edges where each edge, $\vec{e} \in E$,

represents an entity call (u_{caller}, v_{callee}). In-degree of node v, denoted by $deg^-(v)$, is the total number of edges incoming to node v. The out-degree of a node v, denoted by $deg^+(v)$, is the total number of edges outgoing from node v. Figure 3 depicts a Java code snippet with its corresponding call graph.

In OOP, call graph is usually used to represent a system in three different levels of abstraction: Function Call Graph (FCG), Class Call Graph (CCG), and Package Call Graph (PCG). Each graph represents a different view of the system. They are described as the following :

- ***Function Call Graph.*** Represents the low level of the system by capturing the function calls. It can be constructed from the caller-callee relationships. For example, if function v calls function u, then they are represented in a function call graph as two nodes with a directed edge from v to u.

- ***Class Call Graph.*** This graph captures communication between classes, and it represents a coarse-grained function call graph. For example, given a function in class A that calls a function in class B, then the class call graph will contain node A and node B with a directed edge from A to B.

- ***Package Call Graph.*** It represents a coarse-grained class call graph. Classes belong to the same package will be merged as a single node that has the package name, and parallel edges will be removed.

In this dissertation, we suggest more than three levels. The coarsening could be further improved by using the package hierarchy to coarsen the graph to more than the

three levels.

2.3 Related Work

In software engineering research, the study of call graphs, their uses, and functionality have a long history [12, 15, 94]. Several recent works have attempted to use call graphs to assist software engineers in various phases of software development, such as testing, maintenance, and understanding of software evolution [14, 128, 134].

2.3.1 Software Modelling

There are many researches in the software visualization area that depict the structure of the system [17, 89, 119]. Although our visualization tool is mainly developed for visual exploration of the call graph, it can visualize the system structure in different levels of abstraction. Several researchers have proposed different visualization tools to help developers gain a better understanding of the software structure [4,17,66,136]. For example, Alnabhan et al. [4] proposed a 2D software visualization approach. They used geometric forms to represent different entities of the source code. For example, classes and methods are described as rectangles and circles, respectively, and relationships between the methods are represented by arrows. Displaying all system entities, classes, methods, and attributes are likely to lead to an overload of information and failing to adequately represent the software.

Other interesting works exist in this area, including representing a subject program in 3D. Wettel et al. [136] and Brito et al. [17] used a city metaphor, where classes or interfaces are represented as buildings and packages are described as districts. The height,

width, and color of a building reflect different software matrices such as the taller the building, the higher the number of methods. Similar work by Khalooas et al. [66] used a room to represent one class, and rooms are grouped together based on the code's project file structure. Although these tools have been proved to aid developers to comprehend the structure of a system, the dependency among the methods and the classes showing the flow of events in the software system is missing.

2.3.2 Call Graph Visualization

While there are several existing tools that generate call graphs of a system [42, 59, 62, 134]. However, these tool use Graphviz [34, 40] for drawing graphs. For example, Alnabhan et al. [4] proposed a 2D software visualization approach. They used geometric forms to represent different entities of the source code. For example, classes and methods are described as rectangles and circles, respectively, and relationships between the methods are represented by arrows. The generated call graph is represented in a static format such as SVG, PNG, and DOT format. Graphs can get complex when analyzing large systems. Thus, graph entities such as nodes, edges, and associated attributes will likely be overlapping. which requires more effort to understand. Also, the user can not move a node to resolve the overlapping.

Code2graph [42] is a Python static analysis tool that analyze the source code of programs written in Python. The tool can automatically analyze and extract the system structure. Then, construct the static call graph of the system. Finally, construct similarity matrix of all possible execution paths in the system. The tool renders the call graph using

the Graphviz library.

GraphEvo [134] is static analysis tool that captures software evolution using call graphs. The tool mainly focused on finding the differences among subsequent call graphs of different versions of the system and colored the added and removed nodes in Green and Red, respectively, to illustrate the code changes for programs written in Java. The tool also visualize the call graph using the Graphviz library.

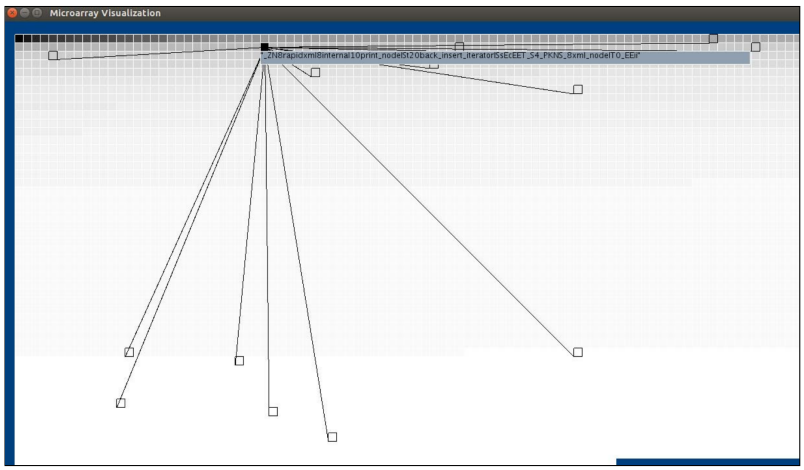


Figure 4: Micro-array Visualization

Other interesting works exist in this area, including interactive visualization for the call graph. Shah and Guyer [119] propose an interactive call-graph visualization tool for programs written in Java or C++. Their tool represents the call graph in a grid of pixels where each pixel or cell represents a function. Figure 4 shows the call graph of a system that represented in micro-array. The microarray layout is inspired by a DNA microarray visualization from biology work proposed by Zhang et. al [145].

REACHER [73] is interactive visualization tool implemented as eclipse plugin. It helps developers explore static call graph paths of the specific method instead of manually

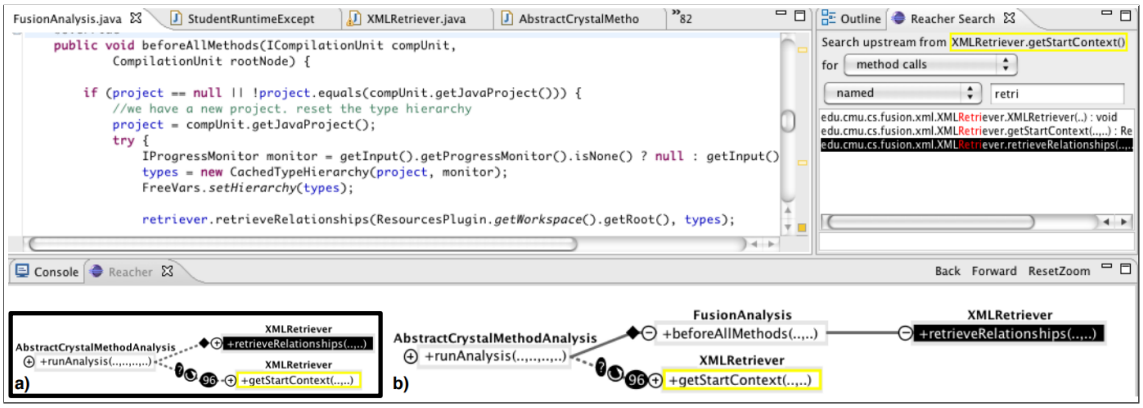


Figure 5: REACHER's Call Graph Visualization

traversing calls to understand the control flow. The tool was developed to help answering a reachability questions such as whether certain code is reachable from other code under certain conditions. Figure 5 shows the two view components of the tool. The search view in the upper right, and the call graph view in bottom.

Another tool that is very similar to ours is introduced by lemos et al. [74]. They proposed SysGraph4AJ (MultiLevel System Graphs for AspectJ). The tool supports visualization of the system's structure and structural testing at the unit level. Although SysGraph4AJ allows users to explore the system in multi-level views, only dependency among the methods is supported.

Bohnet and Döllner [15] combine the static structure and dynamic analysis properties in 3D landscape views. The tool extracts dynamic call graph information and allows the developer to navigate and gain insight into how features are implemented. The dynamic call graph would require the developer to know what inputs data to provide, and how to execute the system. Moreover, obtaining all possible execution paths of the system is very hard. In contrast, our work focuses on the static call graph of the software system.

The static call graph considers all possible execution paths of the system.

These tools are promising but limited in several aspects. (1) limited to a single level of abstraction (i.e., function level) and thus may not well support different maintenance tasks requiring understanding either fine or coarse grain, or both levels. (2) visualize a portion of the call graph and does not depict the overall structure of the system. (3) draw the call graph in a static format which requires more effort to understand. Table 1 summarizes and compares the related work, in addition to our tool CodEx based on six characteristics explained as follows:

- **Abstraction.** Refer to the level of abstraction that the tool support to explore the call graph. low level (e.g., function call graph) or more abstraction levels (e.g., class and package level)
- **Export.** Whether the tool supports exporting the graph data or not.
- **Visualization.** Refer to the the type of the visualization that is used to render the call graph. Static (e.g., DOT, PNG), interactive (e.g., allow clicking, moving the entities, etc)
- **Language.** Programming languages that the tool support.
- **Search and Filter.** Whether the tool supports the searching feature or filtering the graph nodes.
- **Coverage.** Whether the tool supports exploring the complete call graph of a system , or part of it.

Table 1: Tool Features for Call Graph Visualization

Feature/Tool	Abstraction	Export	Visualization	Language	Search and Filter	Coverage
Code2Graph [42]	function level	DOT	Static	Python	None	complete call graph
SysGraph4AJ [74]	three-level	None	interactive	C++, Java	Support	subgraph
GraphEvo [134]	function level	DOT	Static	Java	None	complete call graph
Microarray [119]	function level	DOT	interactive	C, C++, Java	Support	complete call graph
REACHER [73]	function level	None	interactive	Java	Support	subgraph
CodEx	multi-level	GML, Json	interactive	Language independence	Support	complete and sub call graph

2.4 Proposed Approach

Our approach consists of five steps. First, The input of our approach is a source file of the system. Second, we statically analyze the codebase to extract the functions' relationships. Third, the functions' relationships are used to construct the call graph. Finally, visualizing the call graph. Figure 6 depicts the workflow process of constructing and visualizing a call graph of a system.

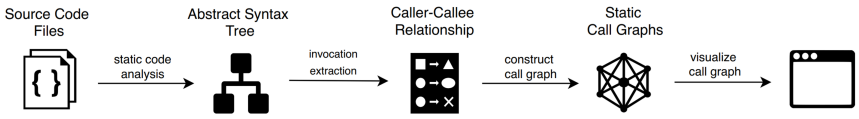


Figure 6: Constructing and Visualizing a Call Graph

2.4.1 Source Code Analysis

The first two components in our approach can be done by using an existing analysis tool. For Java systems, we extract the objects, functions, and their dependencies by integrating an existing static analysis tool, java-callgraph [44] to generate the caller-callee list as a text file. The analysis tool uses the Apache Byte Code Engineering Library (BCEL) to analyze a given $.jar$ file as input.

For Python systems, While there exist many tools to extract entities and their dependencies for several programming languages, such as C, C++, and Java, the methods and tools that address extraction of entities and their dependencies for Python are scarce and limited in functionality, due to the dynamic nature of Python.

Pyan [59] is an open-source project hosted on GitHub that can automatically generate statics call graphs for one or more Python modules. Similar to the majority of tools in this area, Pyan generates DOT files, i.e. graph description language files with the dot extension, which can be rendered into actual graphs using other visualization tools, such as GraphViz. Pyan uses the Abstract Syntax Trees (AST) to process trees of the Python abstract syntax grammar. Using AST helps laying out the codebase structure and its interactions programmatically. Pyan provides a command-line interface to facilitate its functionality and can produce self-organized and colored call graphs. Pyan functionality is limited to a collection of modules inside a single package at a time. However, often, large open-source projects are structured into multiple hierarchical packages, where some packages may include modules and other packages, too. In such a case, Pyan can only process the Python modules at one level of the hierarchy, i.e., it ignores the nested packages and their modules. This enforces the developer to construct call graphs for every single package in the system and then manually relate them to each other. This is tedious and error-prone. Moreover, Pyan does not support the inter-procedural flow of functions leading to missing edges to the parameter functions.

Another Python static analysis tool, named PyCG [114], has been published recently. It resolves some challenges or limitations of Pyan. For example, Pyan does not

support some Python's functionality such as generators, exceptions, and ignores the inter-procedural flow of functions, parameters, and returns.

ENRE [62] is a framework that supports extracting entities and relations from Python and Golang programming languages, and it can be extended to support different programming languages. ENRE uses Antlr [100] as the underlying parser that supports lexical and syntactic analysis of source code. Antlr generates Parse Trees using the grammar rules of a given language. ENRE supports different formats to output entities and relations such as CSV, XML, and JSON. Comparing to PyCG, ENRE supports dynamic relations that can only be precisely resolved at run-time. However, it failed to resolve some cases of relative imports such as $from\ ..A\ import\ B$.

The three aforementioned static analysis tools fail to address all challenges leaving opportunities for improvement. Thus, to get the benefit from each one of the analysis tools, we extend these tools and output the generated caller-callee list in a unified representation. Then, merge them into a single file. Later, we parse the file to construct call graphs of a system.

2.4.2 Unified Representation of Caller-Callee

```
M:sh3d.applet.Applet:isModified() M:sh3d.model.HomeApplication:getHomes()
M:sh3d.applet.Applet:isModified() M:sh3d.model.Home:isModified()
M:sh3d.applet.SweetHome3DApplet:init() M:sh3d.applet.SH3DApplet:createAppletApp()
M:sh3d.HomeFrameController:displayView() M:sh3d.HomeFramePane:displayView()
M:sh3d.io.AutoRecoveryManager:openRecoveredHomes() M:sh3d.model.Home:getName()
```

Figure 7: Part of Unified Representation of Caller-Callee List for SweetHome3D

The first step towards constructing the call graph is to define and extract entities

and their dependencies. We define functions and objects as entities because considering only functions will ignore the object-oriented structure of the system [96]. Also, they are regarded as essential components of traditional systems and represent the functionality of a system more clearly than other components. Relationships between these entities are function calls and class instantiating. To extract the objects, functions, and their dependencies, we extended several existing static analysis tools; java-callgraph [44] for Java, and Pyan, PyCG, ENRE for Python. We output the caller-callee list in a unified representation. Each edge in the caller-callee list is represented using the below format :

```
Flavor:Namespace:Identifier(Parameters)
```

- **Flavor** represents the entity type, such as method, function, or object.
- **Namespace** is the fully-qualified name of the entity, which consists of the package name followed by the class name. It is used to define the scope of the identifier.
- **Identifier** represents the name of the entity.
- **Parameters** are the entity arguments.

Figure 7 shows portion of the methods invocations for one of the case study (i.e., SweetHome3D) using unified representation.

2.4.3 Call Graph Construction

This phase consists of two main steps: (1) parsing the edges list to construct a function call graph, then (2) simplifying the function call graph to different levels of abstraction using the coarsening technique.

2.4.3.1 Graph Schema

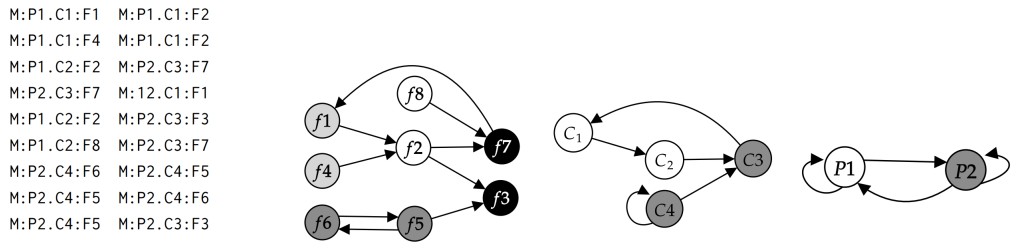
While parsing the caller-callee list, we also extract nodes' properties and their relationships using a generic graph data structure, NetworkX [48]. This library was used to manipulate and render the structure of the call graph. Then, we created a JSON schema to obtain node properties. After constructing the call graph, the results are saved in different file formats, including GML and JSON. JSON schema is used for visualization of the call graph. To avoid the entity name conflict problem, we maintain the fully qualified name of each entity, which consists of the namespace followed by its name and parameters. This is important for two reasons:(1) avoid the method-name conflict problem, since file, classes, and method may have identical names in different namespaces and packages. (2) construct a coarse-grained system call graphs using the namespace of methods. Each entity has the following attributes:

```
{
"idx": "i"
"id": "fully qualified name of the entity"
"namespace" : "PackageName.ClassName",
"label" : "entity name",
"flavor" : "function/object/class/package"
"successors" : []
"predecessors" : []
}
```

More attributes are added to the node after constructing the call graph, such as identifying

the articulation nodes in the graph [58]. An articulation node is a node that disconnects the graph if removed with all its edges.

2.4.3.2 Coarse-grained Representations



(a) Caller Callee List (b) Function Call Graph (c) Class Call graph (d) Package Call graph

Figure 8: Constructing and Visualizing a Call Graph

Coarsening is a popular type of graph reduction, which can be defined as an aggregation process of graph nodes to identify nodes of the next coarser graph [24]. An advantage of a multi-level coarsening technique is to facilitate graph data analysis by merging nodes and edges using specific criteria. In this dissertation, we define our own criteria to simplify the function call graph into multi-level graphs such as Package and Class call graphs. Figure 3 illustrates a toy example showing the coarsening technique. For example, to construct the class call graph, our graph coarsening technique takes the function call graph as an input and then collapses nodes that have the same namespace into a single node. Similarly, we construct the package call graph by merging class nodes with the same namespace, from the class call graph, into a single package node. For large systems, the coarsen will go beyond package call graph by using the package hierarchy to coarsen the graph to more than the three levels.

Algorithm 1 Graph Coarsening

```
1: G      \\ function call graph
2: high   \\ longest namespace
3: low    \\ shortest namespace
4: procedure COARSENING( G, high, low )
5:     G' ← G
6:     while high > low do
7:         high = high − 1
8:         G'[level] ← high
9:         for each node ∈ G' do
10:            level ← countDots(node[namespace])
11:            if level! = 0 then
12:                name ← get the substring after last dot in node[namespace]
13:                namespace ← remove substring after last dot in node[namespace]
14:            node[name] ← name
15:            node[namespace] ← namespace
16:        saveGraph(G')
```

2.4.4 Call Graph Visualization

To implement our approach, We develop an interactive exploratory call graph visualization tool, named CodEx, that can help developers to explore and navigate the call graph of a system written in Java. However, The tool can visualize the call graph of any system written in any programming language. To do this, the user would require any programming language analysis tool and output the method invocation in the unified representation (see Section 2.4.2)

CodEx presents a new set of features that allow the user to interactively explore the call graph in different views, with a side panel to facilitate exploring metadata of the graph and its nodes. We describe these features and demonstrate them with examples

Web-based User Interface. In a website application, a user can open multiple

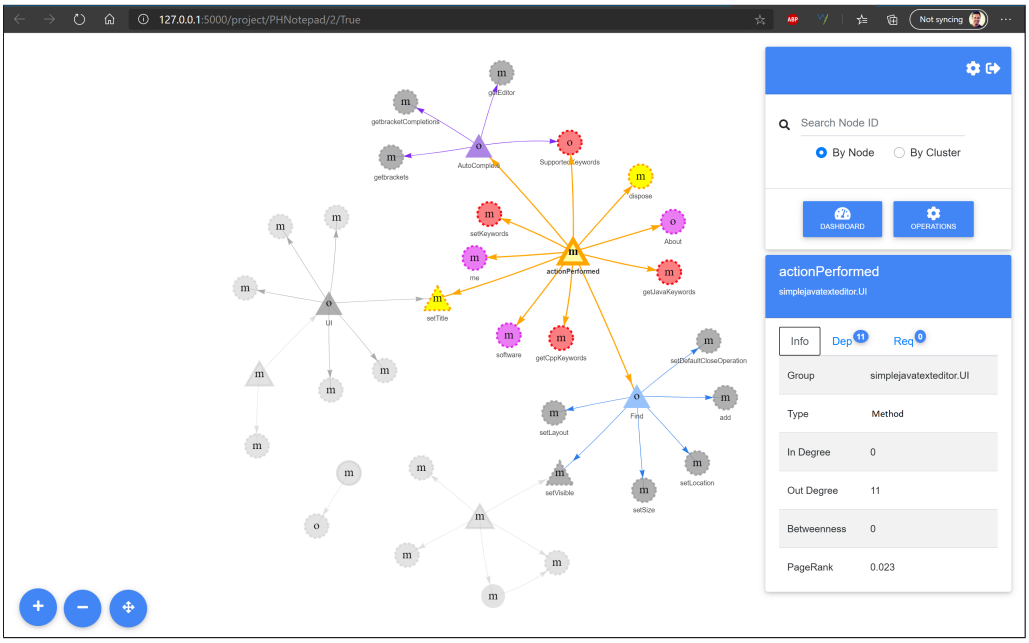


Figure 9: Main Interface

tabs side-by-side in the browser to perform any number of tasks. CodEx utilizes this feature to display different graph levels and views in separate tabs in a browser to help developers studying each level or view separately.

Nodes Shapes and Colors. Node-link technique is commonly used to visualize and explore relationships between software system entities [87]. Thus, we used the node-link technique to represent the call graph of a system. Graph nodes are code coloring based on namespace. For example, in the case of the function call graph, if two nodes belong to the same class or file they will assign for the same color. This can help the developer see how functions/classes of different files/packages interact with each other. We use the node's border to identify whether this node entry or exit point. In particular, if a node has a dashed border then it is an exit point. Similarly, if a node has a bold border

then this node is an entry point. Another node feature is its shape. We use a circle shape to represent an entity, and arrows represent relationships between these entities. The triangle node represents an articulation point, which means removing this node can disconnect the graph into two or more sub-graphs.

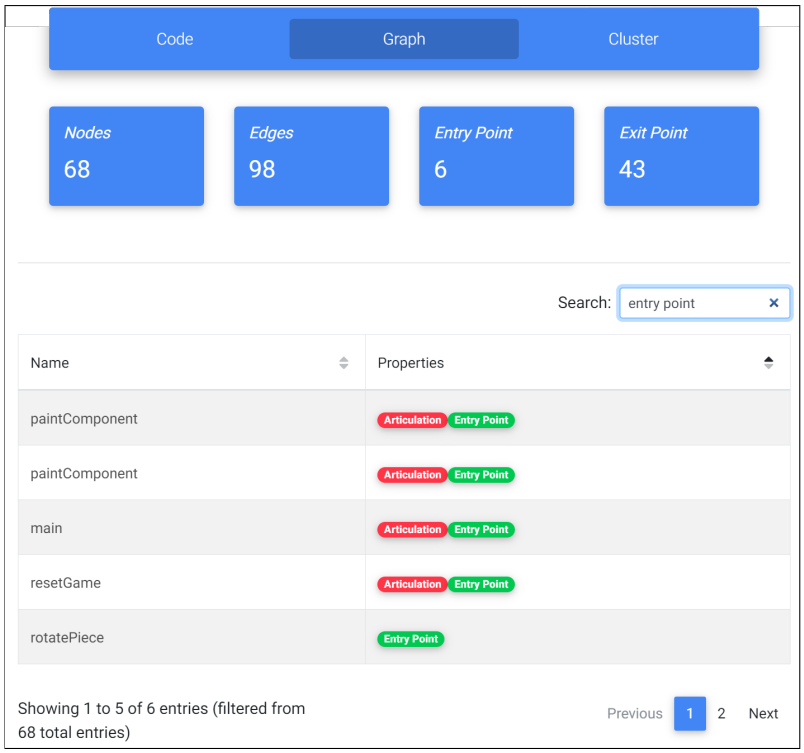


Figure 10: Mini Dashboard

Highlighting Nodes. When the user selects a node by clicking it, CodEx highlights the selected node and its neighbors while coloring the rest of the graph with a transparent gray color. This feature helps developers to focus on particle functions and their relationships. Moreover, selecting a node opens a side panel showing the metadata associated with the selected node (see Figure 9)

The user can easily browse the metadata, including type, in-degree, out-degree,

and graph matrices. Also, clicking on the 'Dependant' tab will show all the node neighbors while clicking on the 'Required' tab will show all nodes that call the current function. As shown in the Figure 9, the user clicked on 'actionPerformed' node/function. From the side panel, we can see that The node calls/depends on 11 nodes while it is called by 0 nodes.

Search and Filtering. Figure 9 shows modal with statistic information about the system and its call graphs, such as the number of nodes, edges, entry points, and exit points. In addition, a table list all the nodes in the graph with their properties. an advanced search, and filter features allowing user to search for nodes with specific property. Figure 10 shows 5 out of 6 entry points in the current graph. This modal can be viewed when the user clicks on the dashboard button in graph view (see Figure 9).

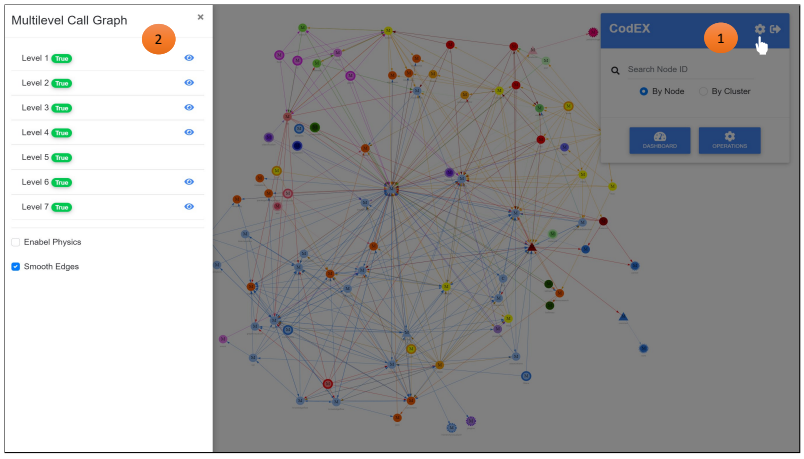


Figure 11: Navigate Between Levels

Navigate between Levels. Figure 11 shows side modal with different levels. Each level represents a call graph with different abstraction. As shown in Figure 11, the system has seven different abstraction levels. Level 7 represents the lowest or finer-grained graph

(i.e., function call graph) while the level 1 represent the most abstract level of the system call graph.

Saving and Loading. Our visualization tool can save the results and state of each process in different formats (e.g., GML, JSON, CSV). Moreover, When a call graph is rendered in the first time, the position of the nodes and edges in the graphs will be saved. Thus, when the user opens the project again the graphs will be loaded in seconds.

2.5 Evaluation

To illustrate and evaluate the utility of our tool to program comprehension, we first conduct six case studies to illustrate that our approach and tool are supporting language-independence. Second, the tool is capable of automatically constructing and visualizing static call graph of a system in multi-levels of abstraction. Third, we present a quantitative evaluation of the case studies to assess the cost and efficiency of our tool using real-world applications. The characteristics of these systems described in Table 2.

Table 2: Subject Systems

System	Version	Description	Language	LOC
Detectron	2.0	Object Detection Library	Python	11,735
Flask	1.1.2	Web Applications Framework	Python	3,315
Keras	2.3.0	Neural-network Library	Python	24,620
PHNotepad	3.0	Code editor	Java	962
SweetHome 3D	6.2	Interior 2D design	Java	104,098
WEKA	3.8	Machine Learning Library	Java	324,497

2.5.1 Case Studies

These case studies are Python and Java open-source projects. The first three case studies are Python projects, namely Detectron [43], Flask [46], and Keras [25], selected based on their ranking on GitHub, i.e., the most popular projects. Detectron is an open-source software system from Facebook that implements state-of-the-art object detection algorithms. It is written in Python and has a relatively small-size codebase. Flask is an open-source web micro-framework written in Python. It provides tools, libraries, and technologies to facilitate web development. Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow [1]. Keras was developed with a focus on enabling fast experimentation.

The remaining case studies are Java projects. PHNotepad [56] is a code editor equipped with operational features such as search, auto-completion, and intuitive, easy-to-use GUI. Although the program system has 962 lines of code, we selected this system due to its outstanding design and manageable size for manual analysis and verification. SweetHome3D [35] is a medium-sized, interior design application. The user can design a house in 2D by drawing the plan of the house, adding furniture and home appliance, and then preview it in 3D. It comes with many easy-to-use features, including import texture and furniture, recording videos, and allowing user-customized preferences, such as setting the system language, fonts, and units of measurement. WEKA [39] is open-source software that provides a set of tools for data mining tasks including data preprocessing, classification, regression, clustering, association rules, and visualization. It helps to develop machine learning techniques and apply them to real-world data mining problems.

Table 3: Structure Analysis Results of the Function Call Graph for Each Case Study

Entity	Detectron	Flask	Keras	PHNotepad	SweetHome 3D	WEKA
Nodes	525	370	1,779	36	5,148	14,742
Edges	740	360	2,347	36	9,501	35,575
Entry Point	108	123	844	4	1,517	5,031
Exit Point	207	168	591	28	2,821	4,982
Articulation Point	133	115	376	8	577	2,460
Coarsen Levels	5	5	5	2	3	7

To evaluate CodEx, we first removed the test cases from each case study since they do not contribute to the overall functionality of the system. Second, For Java projects, our tool is already integrated with the java analyzer. Thus, the developer can use the jar file of the system directly as input for our tool. To construct and visualize a system written in other languages, users have to output the caller-callee list in a specific representation. Thus, for Python projects, we extend three analysis tools PyCG, Pyan, and ENRE to output the method invocations in a unified representation (see Section 2.4.2). Then, save this output as a text file and use it as input for CodEx.

While the main task was to construct and visualize the call graphs, we first illustrate that CodEx is capable of extracting useful statistics metadata about the call graph of the system, presented in Table 3, which can be helpful in understanding the overall complexity of the system call graph.

The call graphs of the case studies showed that Detectron has 525 nodes, out of them is 108 functions are entry points and 207 functions are exit points. The call graph of Flask consists of 370 nodes, out of them is 123 functions are entry points and 168

functions are exit points. We observed that the number of edges 360 relatively close to the number of nodes. The Keras case study was the largest in size among the two other python case studies. In particular, Keras's case study consists of 1,779 nodes, 2,347 edges. It has 844 entry points and 591 exit points. For Java case studies, we see that the smallest case study, PHNotepad, has a similar number of nodes and edges (i.e., 36), 28 of the nodes are exit posits. SweetHome3D consists of 5,148 nodes, 29% of functions are entry points while around 55% are exit points. The WEKA case study is the largest in size among all case studies. It consists of 14,742 nodes, 35,575 edges. It has 5,031 entry points and 4,982 exit points.

2.5.2 Quantitative Evaluation

In order to evaluate the computational efficiency and scalability of CodEx, we conducted the following quantitative evaluation. Specifically, we computed the time cost for constructing a multi-level call graph for each system. In particular, we want to measure the time is needed to construct the low-level graph (i.e., function call graph) until the most abstract call graph using our coarsening technique for each case study. The results are summarized in Table 4.

Table 4: Summary of the Quantitative Analysis Results

Entity	Detectron	Flask	Keras	PHNotepad	SweetHome 3D	WEKA
No. Levels	5	5	5	2	3	7
Coarsen Process	0.874	0.757	1.438	0.127	4.426	18.397

2.6 Conclusion

In this chapter, we show how a system's call graph can be used as a starting point for program comprehension by developing presented an automated language-independent approach for analyzing the source code of a system and visualizing its structure in multi-level abstractions, equipped with an interactive visualization tool. To illustrate and evaluate the utility of our tool to program comprehension, we conduct six case studies to illustrate that our approach and tool are (1) supporting language-independent, (2) capable of automatically constructing and visualizing static call graph of a system in multi-levels of abstraction, and (3) we present a quantitative evaluation of the case studies to assess the cost and efficiency of our tool using real-world applications.

CHAPTER 3

STATIC TRACE CLUSTERING: SINGLE-LEVEL APPROACH

3.1 Introduction

As the system evolves over time and its complexity increases, a larger amount of data, including source code, bug reports, test cases, execution paths, etc, can be generated. Mining and analyzing software data can give further insight to support decision-making related to the software development life cycle (SDLC). Especially, during software development and software maintenance.

A program's static execution path is defined as the sequence of method calls. These calls can be represented as a graph with nodes representing functions and directed edges representing calls between these functions. The graph derived from a static trace is known as a static call graph. A static call graph represents the system's functions and their interactions. However, with the increasing size and complexity of software systems, the generated call graphs are typically very large, including thousands of functions and millions of execution paths. Such large call graphs can have a contrary effect on software comprehension; they could be overwhelming and more difficult to understand. In addition, the call graph is limited to a single level of granularity, such as function calls. Developers often need to manually map their understanding of the overall functionality of the system to its low-level implementation captured by the call graph. Manually mapping the high-level functionality of the system to its low-level implementation is expensive,

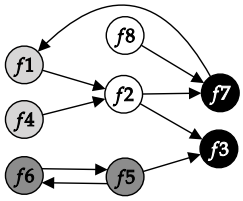


Figure 12: Function Call Graph

time-consuming, and error-prone. Therefore, there exists a cognitive gap between the high-level functionality of a software system and its low-level implementation.

Many of the execution paths perform similar functionalities and share a big number of functions with other paths. Therefore, clustering can play a key role in creating hierarchical abstractions from the low-level function calls to the high-level system architecture. For example, in Figure 12, two of the execution paths: $F1 \rightarrow F2 \rightarrow F3$ and $F4 \rightarrow F2 \rightarrow F3$ are considered similar and can be grouped in one cluster although they are not identical. Iteratively, clustering similar execution paths can bridge the gap between the source code and the overall system functionality. Thus, helps developers understand the system functionality at several hierarchical levels of abstraction.

In order to overcome the issue of representing the system's call graph at a single level of granularity, researchers have proposed approaches that can represent the call graph in multiple abstraction levels [38,41,143]. This can help developers understand the system and investigate its functionality at several levels of abstraction. However, some of these techniques cluster the call graphs based on the dynamic execution of the program, the package structures, or consider only a few execution paths of the software system. Moreover, little attention has been given to the area of visually examining and interpreting

these clusters in terms of the call graph. The mentioned clustering approaches visualize clustering results using flat lists or static hierarchical dendrogram [41]. Which makes it difficult for users to explore, interpret, and navigate.

This chapter describes a new automatic data-driven technique to extract static traces of a system and hierarchically abstract these static traces into a multi-level of granularity.

3.2 Overview of Hierarchical Clustering

Clustering is the process of grouping entities together based on their properties (i.e., features). It has been applied in many fields such as bioinformatics, image segmentation, document retrieval, and financial analysis [7, 79, 101], where similar objects are grouped together. Thus, making large datasets more comprehensible. The clustering approach can be broadly classified into two categories: hierarchical and partitional. Hierarchical clustering consists of a sequence of nested data partitions in a hierarchical tree while partitional clustering produces flat clusters with no hierarchy, and requires prior knowledge of the number of clusters. A popular example of partitional clustering methods is the K-means algorithm. It is known as one of the most popular partitional clustering methods [11, 61]. It was actually recognized as one of the top ten algorithms for data mining [142].

For hierarchical clustering methods, there are two hierarchical clustering approaches, Divisive clustering (top-down) and agglomerative clustering (bottom-up). We will discuss and explore hierarchical agglomerative clustering algorithms, since it is widely used for

modularization and architecture recovery of software systems [82].

Agglomerative Clustering. is one of the most common methods used in software design. The reason is due to the very similar nature of its work to reverse engineering process, where abstractions of software design are recovered in a bottom-up manner [26]. As shown in algorithm 2. The algorithm starts by considering each data point as a singleton cluster. Then, it merges the two most similar clusters until it forms a single large cluster.

Algorithm 2 Agglomerative Clustering Algorithm

```
1: procedure CLUSTERING(dataset)
2:     Compute the similarity matrix
3:     Let each data point in dataset be a cluster
4:     repeat
5:         Merge the most similar cluster based on linkage criteria
6:         Update the similarity matrix
7:
8:     until single cluster remains
```

Agglomerative Clustering does not require specifying any particular number of clusters. However, it has different approaches (i.e., linkage) to specify how exactly the "most similar cluster" is measured. The following are the main types of linkages:

- **Single Linkage.** Also knows as the nearest neighbor. It is defined as the shortest distance between a pair of data points in two clusters. Formally, the distance between two clusters can be identified by taking the nearest point i in cluster C_1 from the closest distant point j in cluster C_2 as shown in the following equation.

$$d(C_1, C_2) = \min(d(C_1[i], C_2[j])) \tag{1}$$

- **Avarage Linkage.** The average distance between each data point in one cluster to

every point in the other cluster.

$$d(C_1, C_2) = \sum_{ij} \frac{d(C_1[i], C_2[j])}{(|C_1| * |C_2|)} \tag{2}$$

- **Complete Linkage.** Also knows as the farthest neighbor. It is defined as the longest distance between a pair of data points in two clusters. Formally, the distance between two clusters can be identified by taking the farthest point i in cluster C_1 from the most distant point j in cluster C_2 as shown in the following equation.

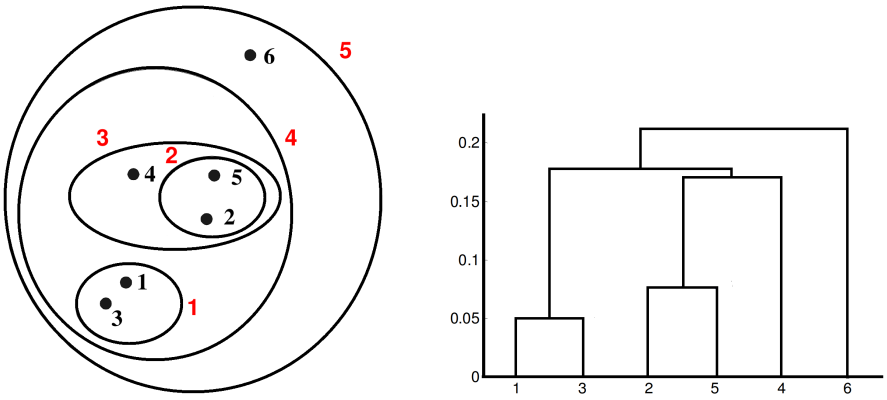
$$d(C_1, C_2) = \max(d(C_1[i], C_2[j])) \tag{3}$$

Figure (13.a) illustrates the progression of the agglomerative cluster. Initially, each point is a singleton cluster. Then, in each step, the two closest clusters are merged. In the first step, clusters 1 and 3 are picked and these are joined into two-point clusters. In the second step, clusters 2 and 5 merge into one cluster. In step 3, cluster (2-5) is extended to a third point, and so on. In step 5, there is only one large cluster remaining. Thus, the algorithm then stops. As shown in Figure (13.b), the progression of the agglomerative cluster can be also represented as tree-like (known as Dendrogram).

3.3 Related Work

3.3.1 Software Clustering

Software clustering is one of the commonly used techniques for program comprehension. Since source code is the most reliable and important input for program comprehension, Several clustering approaches use the source code as the only trusted source to



(a) Nested Clusters (b) Dendrogram

Figure 13: Agglomerative Clustering

understand the software. Software clustering approaches can be divided into two different categories: syntactic and semantic. The syntactic approaches are based on static structural dependencies (e.g., inheritance, method calls, references). A number of syntactic-based clustering techniques have been proposed [60, 90].

The semantic approaches consider analyzing and extracting domain knowledge of the system by clustering the source code in terms of comments and identifier names [64, 70, 116, 125]. Unlike previous software clustering techniques, our approach uses static execution paths to guide the clustering of source code entities that perform similar functionality. Execution paths can also reflect the overall system workflow [63].

3.3.2 Clustering Execution Paths

Xin, Qi, et al. [143] proposes $FeatureFinder$, an approach that aims to identify the features of a program by clustering its execution paths using a classifier-based algorithm. However, this approach is limited to a single level of granularity (i.e., function

level). Gharib, et al. [42] introduces $code2graph$, an static analysis approach that clustering execution paths using the Jaccard similarity coefficient. The similarity between all possible system paths is represented using a heat-map. However, in earlier work, Gharib, et al. [41] provided such a trace abstraction technique by clustering execution paths in a hierarchical fashion to find different levels of granularity and visualize those in a dendrogram. Similar to this approach, Feng, Yang, et al [38] presents $SAGA$, an dynamic analysis approach that clusters the execution paths into multiple levels of granularity. Sage visualizes the identified clusters in vertically stacked layers, where users may lose the position of the visible layer due to the lack of a global overview.

Table 5: Comparative Table of Execution Paths Clustering Techniques

Approach		Pros	Cons
SEGA	Dynamic analysis approach that clusters the execution paths into multiple levels of granularity	- Cluster paths into multi-level of granularity - Labeling clusters based on methods names using TF-IDF	- Single level of abstraction (i.e., function level) - Does not consider all possible execution paths - Visualizes clusters in vertically stacked layers, where users may lose the position of the visible layer due to the lack of a global overview
Code2Graph	Static analysis approach that clusters the execution paths into multiple levels of granularity	- Depict overall structure of the system - Cluster paths into multi-level of granularity - Labeling clusters based on methods names using TF-IDF	- Single level of abstraction (i.e., function level) - Lack interpretation of the abstraction levels - Suffer from scalability issues - Ignore Infeasible path
FeatureFinder	Dynamic analysis approach that cluster execution path using a classifier to decide relatedness	- Labeling clusters based on methods names using TF-IDF	- Does not consider all possible execution paths - Single level of abstraction (i.e., function level) - Lack of hierarchical abstraction - Lack of a global overview

3.4 Proposed Approach

In this section, we introduce our approach for hierarchically abstracting execution paths. The overview of our approach presented In Figure 14. The input of our approach is a call graph of a system, which can be constructed from the method invocation. More details about constructing and visualization the call graph of a system have been introduced in Chapter 2. The call graph is a prerequisite to generate all execution paths of the system. An execution path represents all function calls between an entry point (i.e., function) to an

exit point in the system. After collecting all the paths, we cluster them into a multi-level of granularity using the hierarchical clustering technique.

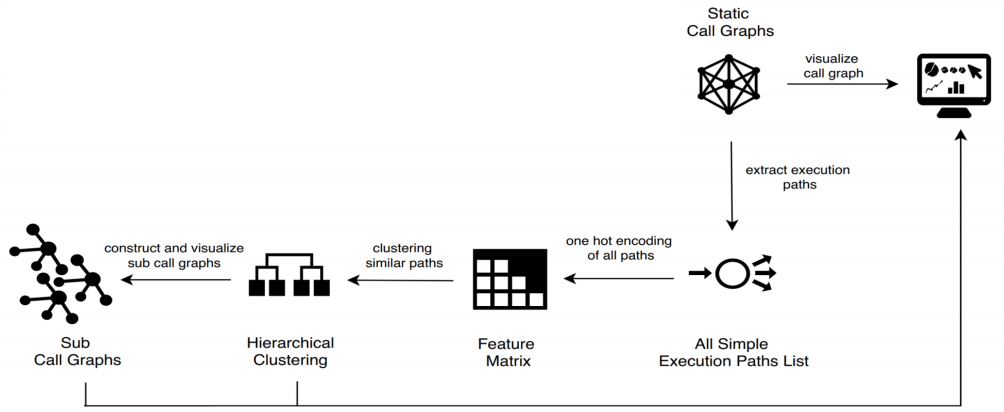


Figure 14: Constructing and Visualizing a Call Graph into Multi-level of Granularity

3.4.1 Execution Paths Extraction

A simple execution path is a list of edges connecting a list of vertices $v_1, v_2, v_3, ..., v_n$ with the restrictions that all edges have the same direction. In addition, none of the edges or vertices can be repeated.

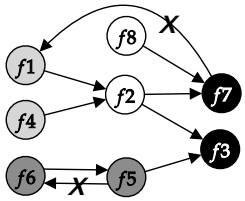


Figure 15: Handling Infeasible Paths by Removing Back Edges.

In this step, we need to extract all possible simple execution paths from each source node s to all target nodes t. Each source node represents an entry point, and each

target node represents an exit point in a system. Based on the number of in-degrees and out-degrees of a node, we can identify the type of the node: A node with $deg^-(u) = 0$ is called an entry point, and a node with $deg^+(u) = 0$ is called an exit point.

A simple execution path of a graph is a path that does not include any cycles. We use all_simple_paths algorithm built-in NetworkX library [48] to extract all simple execution paths. This algorithm uses a depth-first search to generate the paths in the graph between the given entry and exit points. However, before that, we extended the algorithm to break the self-loop to create more entry points and exit points. We also consider breaking back edges in the graph to remove the cycles. Our criteria for breaking cycles rely on the order of back edge discovery. For example, in the Figure (15), the edge $(F6, F5)$ comes before the edge $(F5, F6)$. Thus, the back edge, in this case, is $(F5, F6)$. It will be not considered when we extract all simple execution paths.

Table 6 shows an example of generated simple paths from the function call graph in Figure 15. This process results in a list of size P, where P is the total number of paths. Each path has a list of functions $P_i = [f1, f2, ..., fn]$. All paths are exported to a CSV file where each row represents a path.

Table 6: Example of Execution Paths (P:path, f:function)

P_1	f_1	f_2	f_3
P_2	f_1	f_2	f_7
P_3	f_4	f_2	f_3
P_4	f_4	f_2	f_7
P_5	f_6	f_5	f_3
P_6	f_8	f_7	-

3.4.2 Feature Matrix

Before applying software clustering, we need to identify entities to be clustered where each entity is described by different features. The selection of an entity depends on the objective of the method. For example, for software modularization, researchers used several types of entities, such as files [5], classes [10], and methods [113]. Researchers have also used different types of features to describe these entities such as global variables used by an entity [93], and procedure calls [5].

Our approach uses execution paths as entities, and method calls are the features that describe these entities. Before clustering the execution paths using machine learning techniques, we first preprocess our data and put it in a consumable format. Then, a feature matrix of size $N\ x\ M$ is generated, where N is the total number of entities, and M is the total number of features. In particular, We encode the paths and features in a one-hot encoding manner.

Table 7: Example of Feature Matrix

Entity/Feature	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8
P_1	1	1	1	0	0	0	0	0
P_2	1	1	0	0	0	0	1	0
P_3	0	1	1	1	0	0	0	0
P_4	0	1	0	1	0	0	1	0
P_5	0	0	1	0	1	1	0	0
P_6	0	0	0	0	0	0	1	1

Table 7 illustrates the entities in the execution paths. Each entity in the feature matrix has a feature vector, f_i = [f_1, f_2, . . . , f_n]. These features represent the presence or absence of a function in a given path. Table 7 shows an example of feature matrix, which

contains 6 entities (P_1–P_6) and eight binary features (f_1–f_8). We notice that f_1 is present in entities P_1 and P_2, while absent in the rest of entities (P_3-P_6).

3.4.3 Hierarchical Clustering

Hierarchical clustering is a distance-based algorithm that uses a similarity function to measure the distance between two clusters, i.e., how close they are. It allows the developer to explore the data on different levels of granularity. There are two hierarchical clustering approaches, Divisive clustering (top-down) and agglomerative clustering (bottom-up). Our approach uses agglomerative hierarchical clustering (AHC) to cluster the execution paths into hierarchical abstractions.

As shown in Algorithm 3, each path is a singleton cluster, and then the two most similar clusters are joined at each step until it forms a single large cluster, which contains all the paths. There are several advantages of using AHC for our approach. AHC process is more similar to the reverse engineering approach, where the architecture of a software system is recovered in a bottom-up fashion [139]. Moreover, AHC provides different levels of abstraction and can be useful for developers to select the desired number of clusters when the results are valid and meaningful.

Algorithm 3 Agglomerative Clustering Algorithm

```
1: procedure CLUSTERING( SimMatrix, linkage )
2:     cluster ← {}
3:     for each p in paths do
4:         cluster ← cluster ∪ p
5:     while cluster ≠ 1 do
6:         Join the two closest clusters
7:         Update the distance matrix
```

Hierarchical clustering does not require specifying the number of clusters. However, performing this clustering requires two things: (1) similarity measures of execution paths and (2) the linkage type between two clusters. Several researchers conducted experiments on a set of systems to compare various similarity measures and linkage types [6, 30, 82]. They concluded that Jaccard similarity produces more reliable measurements as well as the complete link algorithm. Thus, we present our results by using the Jaccard as a similarity measure and complete link type.

3.4.3.1 Similarity Measures

Similarity metrics compute a coupling value between two entities. To measure the similarity between a pair of entities, we used Jaccard similarity [83], which measures the dissimilarity between two sets. It is widely used in clustering problems, such as text clustering. It can be calculated by computing the size of the intersections divided by the size of the union of two sets and then subtracting the result from one, as shown in the following equation.

$$d(P_i, P_j) = 1 - \frac{|P_i \cap P_j|}{|P_i \cup P_j|} \tag{4}$$

3.4.3.2 The Linkage Type

Hierarchical Agglomerative clustering comes with different variants to measure the distance between two clusters, known as linkages. There are three main types of linkages used in many software architecture recovery techniques [82, 83]. The types include Single Linkage (SL), Complete Linkage (CL), and Average Linkage (AL). We use the complete linkage method in our work to measure the distance between two clusters,

which is identified by taking the farthest point i in cluster C_1 from the most distant point j in cluster C_2 as shown in the following equation.

$$d(C_i, C_j) = \max(d(C_1[i], C_2[j])) \tag{5}$$

3.4.4 Converting Cluster to Graph

Hierarchical clustering produces a dendrogram, which illustrates how the AHC is performed in a bottom-up approach. AHC starts with the low-level execution paths up to the root, where the linkage algorithm is completed. We have different levels of abstractions for a given system in the form of a tree. To get a multi-level granularity of the system call graph, we need first to convert these clusters to graphs. We first extract cluster data from dendrograms, such as paths that belong to each cluster. This process results in a list of size *C*, where *C* is the total number of clusters. Each cluster has a list of paths $Ci = [P1, P2, ..., Pn]$. All clusters are exported to a CSV file, where each row represents a cluster, and each column represents a path ID. Table 8 lists an example of extracted clusters from a given dendrogram example.

Algorithm 4 takes the id of a cluster as input to retrieve all the paths that are part of the cluster. Then each path in the selected cluster will be extracted from the path file. This can be done by pointing the path's id to its corresponding line in the path file. Finally, we pass these paths into G to build a call graph of the cluster. Later, the call graph of the cluster will be mapped to the original call graph using our visualization tool. (see Chapter 6)

Table 8: Example of a Clusters Table (C: cluster, P: path)

C_1	P_1					
C_2	P_2					
C_3	P_3					
C_4	P_4					
C_5	P_5					
C_6	P_6					
C_7	P_1	P_2				
C_8	P_3	P_4				
C_9	P_1	P_2	P_3	P_4		
C_{10}	P_1	P_2	P_3	P_4	P_5	
C_{11}	P_1	P_2	P_3	P_4	P_5	P_6

Algorithm 4 Cluster to Call Graph Conversion

```
1: procedure CLUSTER_TO_GRAPH( cluster_id)
2:     pathsList ← GetClusterPaths(cluster_id)
3:     G ← DiGraph()
4:     for path_id ∈ pathList do
5:         path ← GetPath(path_id)
6:         G.add_path([path])
7:     return G
```

3.5 Evaluation

In order to illustrate and evaluate our clustering approach, we developed a web-based tool, named CodEx (see Chapter 6), and used it to conduct a case study. We first present the case study to illustrate that our approach and tool are capable of constructing and generate meaningful hierarchical clusters at different levels of granularity. Second, we present a quantitative evaluation by conducting six case studies to assess the performance overhead of clustering execution paths.

3.5.1 Case Study: PHNotepad

This section presents a case study to illustrate the applicability and usefulness of our approach. In this case study, we examined if we can generate meaningful hierarchical clusters at different granularity levels. We applied our approach to PHNotepad, a Java code editor written in Java. The software is equipped with operational features such as search, auto-completion, and intuitive, easy-to-use GUI. Although the program system has 962 lines of code, we selected this system due to its outstanding design and manageable size for manual analysis and verification.

Objective. One of the challenges in unsupervised learning algorithms is to evaluate whether the used clustering algorithm produces meaningful results. Thus, our first case study focused on a manageable size subject that enabled us to inspect the results manually and evaluate their significance. Also, we wanted to examine if our approach can identify the functionalities of the system.

3.5.2 Results and Discussion

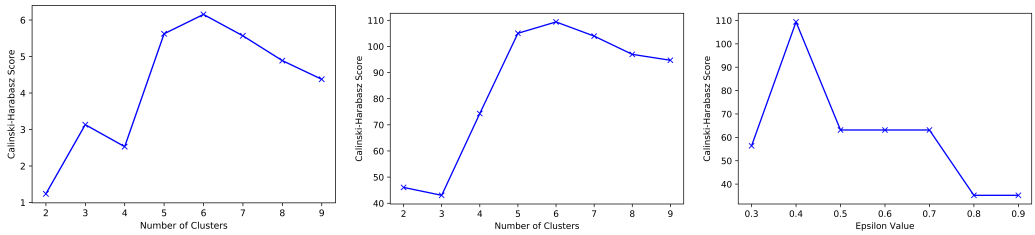
Our tool automatically generates different levels of the call graph for a given software system using the coarsening technique presented in the previous chapter (see Section 2.4.3.2). Since PHNotepad has only one package, the tool automatically ignores the package call graph. Table 9 shows the analysis results for the class and function call graphs. We notice that the system has a single entry point in the class level graph. Using the search feature in our visualization tool, it is noted that the entry point is a class called '$SimpleJavaTextEditor$' that contains the main method calling all the other methods

required to run the application. Since we are interested in the low level, We will ignore the class level and apply our approach to the function call graph.

Table 9: Call Graph of PHNotepad in Class and Function Levels

Entity	Class Call Graph	Function Call Graph
Nodes	6	36
Edges	7	36
Entry Point	1	4
Exit Point	3	28
Paths	7	32

We apply three different clustering techniques (AHC, DBSCAN [49, 50] and K-Mean [53]). The agglomerative hierarchical clustering can capture the overall feature of high dimensional data. However, The other clustering algorithms do not perform well on high dimensional data. Thus, we apply dimensional reduction technique called T-SNE [129] to compute a new low-dimension representation for the data, and compare the results with dendrogram. In order to determine the number of clusters, we use Calinski-Harabasz to computes the optimal number of K in the clustering. Figure (16) presents the scores of different values of K clusters with a range from 2 to 10. The highest Calinski-Harabasz refers to the optimal value of K. Based on the Calinski-Harabasz index, the K value of 6 yielded the highest score. Thus, the functional call graph data were clustered into six groups. Comparing K-Mean to the agglomerative clustering, although both agreed that cluster 6 is the optimal number, k-Means gets the highest CH score. The reason is K-Mean used the new representation of the data which in turn improves the score of CH. For DBSCAN, we set different epsilon values. The optimal epsilon value is 0.4 which gives us 6 clusters too.



(a) AHC (b) K-Mean (c) DBSCAN

Figure 16: Comparing Three Clustering Algorithms on PHNotepad's Paths

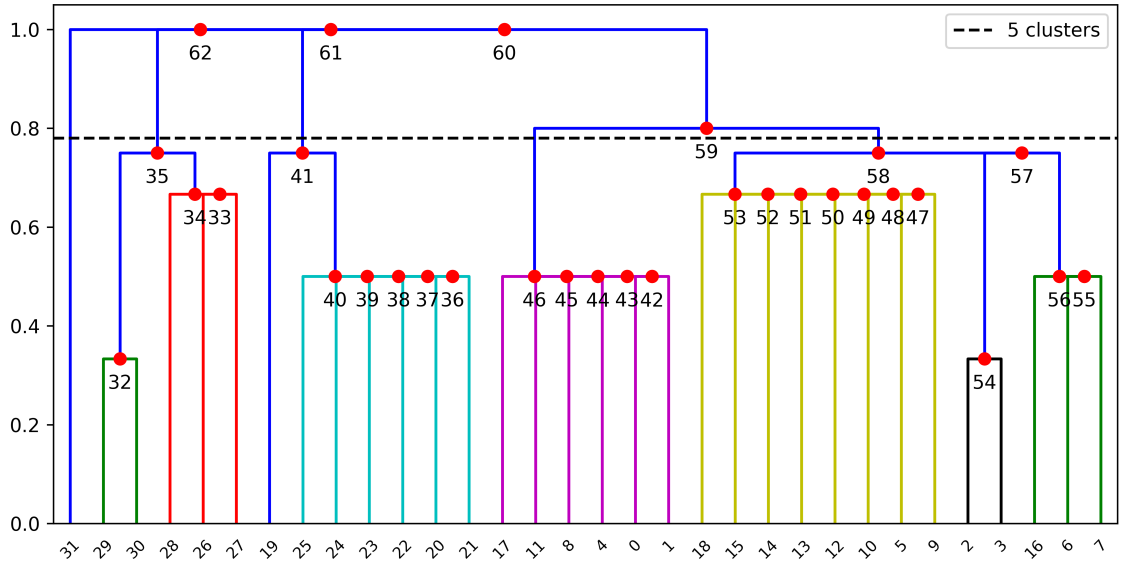
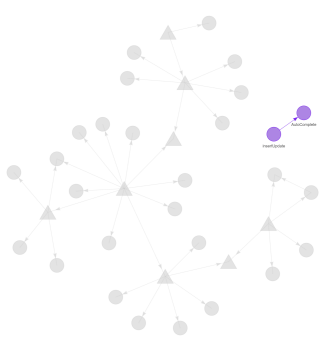


Figure 17: Dendrogram of PHNotepad

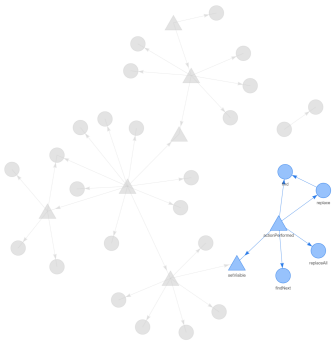
Figure 17 shows the result of hierarchical clustering using dendrogram. The agglomerative hierarchical clustering is reasonably close to the software structure of the system. We notice that Cluster 31 does not group with any other cluster except the last cluster due to the nature of the algorithm, where all clusters eventually will group together. We used our tool to visualize the function call graph of Cluster 31 (Figure 18a). We found that Cluster 31 represents the auto-complete feature for matching brackets. Also, we observed that this cluster represents a disconnected graph, which is shown as a singleton cluster in the dendrogram. With more investigation with the function call graph, we found that these two nodes depend on functions from external libraries, which was not considered during the call graph construction. For Cluster 35, when we mapped it to the call graph (Figure 18b), we found that it represents the search feature in PHNotepad. Cluster 19 (Figure 18c) initially starts as a singleton cluster. This cluster is responsible of setting the GUI components visible. Later, it merges with Cluster 40 due to sharing a node, i.e., the main method. Cluster 40 (Figure 18d) handles the initialization of the main graphical interface of the system.

Our results represent that there are strong relationships between some clusters. Cluster 46 (Figure 18e) handles the graphical interface of the search feature in the system. Cluster 53 (Figure 18f) handles all buttons actions in main interface. Cluster 57 (Figure 18g) handles the auto-complete feature. Cluster 62 (Figure 18h), which is the root, represents the complete call graph without missing any node or edge.

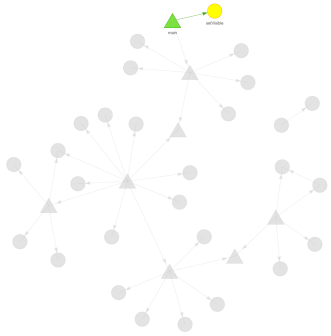
Our intensive evaluation confirms that the call graph results generated correct execution paths of the PHNotepad system. Thus, it is demonstrated that the proposed



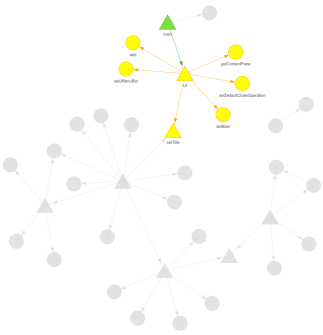
(a) Cluster 31



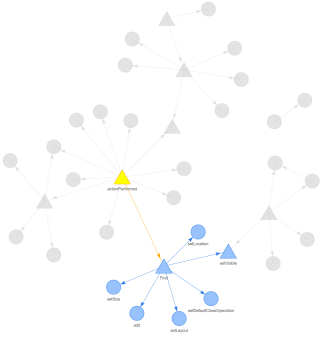
(b) Cluster 35



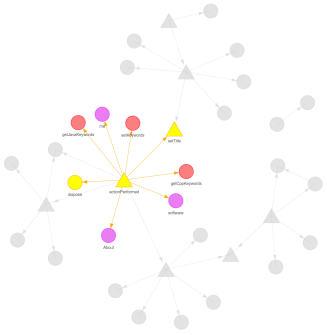
(c) Cluster 19.



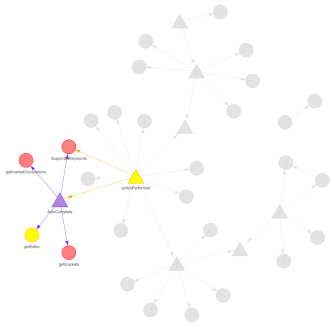
(d) Cluster 40



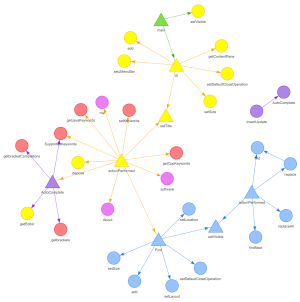
(e) Cluster 46



(f) Cluster 53



(g) Cluster 57



(h) Cluster 62

Figure 18: Converting PHNotepad clusters from the dendrogram to call graphs. (a) Autocompleted matching brackets. (b) Representing the search feature action events. (c) Setting GUI components visible. (d) Providing the Main GUI (e) Handling GUI of the search feature. (f) Handling all buttons' actions in the main interface. (g) Handling the autocomplete programming language keywords feature. (h) Presenting the function call graph of system

methods can be used for building abstraction automatically by extracting call graphs from source file and clustering them. In addition, the visualization tool is very useful for further exploration and comprehension of the automatically generated call graphs. Therefore our approach is capable of extracting, clustering, and visualizing meaningful representations of a given software system with minimal user intervention, which ultimately facilitates and accelerates the overall program comprehension task.

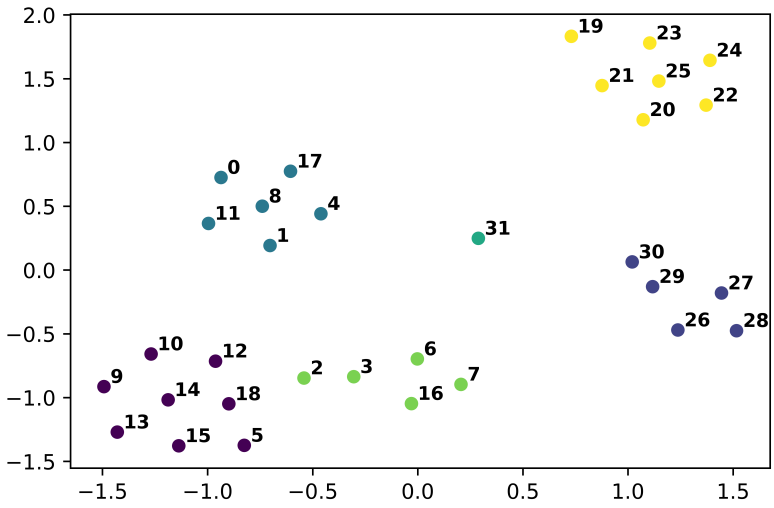


Figure 19: Execution Paths After Reducing Dimensions to 2D Using T-SNE

Figure 19 shows a remarkable result of t-SNE. Cluster assignments with 6 clusters. Both K-Mean and DBSCAN has similar cluster assignments. All the clusters are quite clearly separated. The new representation is reasonably close to dendrogram. We can compare them manually and see how they are very close to their locations in dendrogram. For Example, in Scatter plot, we see point 31 does not belong to any group as well as in dendrogram. Also, point 19 in scatter plot looks a little far from their neighbors which we can identify in dendrogram.

3.5.3 Execution Time Overhead

We measured the time cost for each process that we applied. Our main goal here is to measure the amount of time needed to produce call graphs with the multi-level of granularity. As shown in Table 10, "Hierarchical Clustering" presents the vast majority of the processing overhead at each level. This is due to the high computation complexity of AHC.

Table 10: Time Cost for Each Step in Second

Entity	Detectron	Flask	Keras	PHNotepad	SH3D	WEKA
Call Graph	0.544	0.501	1.118	0.026	0.291	15.996
Execution Paths	6.544	0.950	27.722	0.002	146.390	10169.478
One Hot Encoded	2.525	0.451	17.214	0.001	71.613	9505.750
Hierarchical Clustering	44.841	4.811	93.108	0.686	3211.462	-
Generate Subgraph	0.141	0.102	0.193	0.079	0.459	-
Total	54.595	6.815	139.355	0.794	3430.215	-

By examining the results of PHNotepad shown in Table 10, we observe that the entire processing time is around 0.7 seconds for constructing the function call graph with multi-level of granularity. The longest sub-process time, 0.686 seconds, representing 75.8% of the entire processing time, was spent on the hierarchical clustering process. As for the second most costly process, generate the clustered graph, 0.079 seconds. The time cost of generating all the 63 clusters is 0.189 seconds. However, the cost would be much lower if a specific cluster is selected to investigate.

For Python case studies, namely, Detectron, Flask, and Keras, the overall cost of all processes is around 1 minute, 7 seconds, and 2.3 minutes respectively. By examining the results of SweetHome3D, we observe that the entire cost of all processes is around

57.17 minutes. The longest sub-process time, 53.52 minutes, representing (93.62%) of the overall cost, was taken on the hierarchical clustering process.

By examining the results of the largest in size among all case studies, WEKA. The time cost of construction the function call graph is around 16 seconds. Generating all possible execution paths takes around 2.8 hours while encoding all paths takes 2.6 hours. However, applying the hierarchical clustering process was not successfully completed due to limited memory and computational resources. We address this problem in the next chapter by introducing a scalable approach.

3.6 Conclusion

In this chapter, we develop a data-driven approach of static analyzes to mine a system's execution paths and determine the implementation detail of a system. Many of the execution paths perform similar functionalities and share a huge number of functions with other paths. Therefore, to handle this large volume of execution paths, we apply clustering techniques to group these paths based on similar functionalities they share. The goal of our clustering technique is to bridge the cognitive gap between the system's overall functionality and its implementation by automatically mapping high-level system functionality to its low-level implementation. To evaluate our approach, we conduct a case study using our clustering-based tool, named CodEx. We present the case study to illustrate that our approach and tool are capable of constructing and generate meaningful hierarchical clusters at different levels of granularity. Then, we present a quantitative evaluation of six case studies to assess the performance overhead of the approach

Table 11: All Possible Execution Paths of PHNotepad

ID	Path		
0	UI.actionPerformed	Find.Find	Find.setLayout
1	UI.actionPerformed	Find.Find	Find.setLocation
2	UI.actionPerformed	SupportedKeywords.SupportedKeywords	
3	UI.actionPerformed	AutoComplete.AutoComplete	SupportedKeywords.SupportedKeywords
4	UI.actionPerformed	Find.Find	Find.setVisible
5	UI.actionPerformed	About.me	
6	UI.actionPerformed	AutoComplete.AutoComplete	SupportedKeywords.getbracketCompletions
7	UI.actionPerformed	AutoComplete.AutoComplete	SupportedKeywords.getbrackets
8	UI.actionPerformed	Find.Find	Find.setDefaultCloseOperation
9	UI.actionPerformed	UI.setTitle	
10	UI.actionPerformed	About.About	
11	UI.actionPerformed	Find.Find	Find.add
12	UI.actionPerformed	UI.dispose	
13	UI.actionPerformed	SupportedKeywords.setKeywords	
14	UI.actionPerformed	SupportedKeywords.getJavaKeywords	
15	UI.actionPerformed	SupportedKeywords.getCppKeywords	
16	UI.actionPerformed	AutoComplete.AutoComplete	UI.getEditor
17	UI.actionPerformed	Find.Find	Find.setSize
18	UI.actionPerformed	About.software	
19	SimpleJavaTextEditor.main	UI.setVisible	
20	SimpleJavaTextEditor.main	UI.UI	UI.setJMenuBar
21	SimpleJavaTextEditor.main	UI.UI	UI.add
22	SimpleJavaTextEditor.main	UI.UI	UI.setTitle
23	SimpleJavaTextEditor.main	UI.UI	UI.setDefaultCloseOperation
24	SimpleJavaTextEditor.main	UI.UI	UI.getContentPane
25	SimpleJavaTextEditor.main	UI.UI	UI.setSize
26	Find.actionPerformed	Find.setVisible	
27	Find.actionPerformed	Find.findNext	
28	Find.actionPerformed	Find.replaceAll	
29	Find.actionPerformed	Find.find	
30	Find.actionPerformed	Find.replace	Find.find
31	AutoComplete.insertUpdate	AutoComplete.checkForBracket	

CHAPTER 4

STATIC TRACE CLUSTERING: MULTI-LEVEL APPROACH

4.1 Introduction

In an attempt to discover the feature and structure of a software system, traces clustering has been used in various research works in trace analysis. However, these researches encounter performance overheads introduced as a result of computing the similarities between paths and hierarchically clustering them. Due to this shortcoming, many of the previous works in this area does not scale well. In recent years, a number of trace clustering techniques for program understanding have been published [38, 41, 42, 143]. However, these techniques typically consider a huge number of low-level information (e.g., methods invocation) that may not be needed in a certain domain. Unfortunately, applying a variety of computational analysis techniques and understanding these low-level paths is a time-consuming, performance overhead and does not scale well for large systems. To alleviate the overhead we adopt the multi-level graph partitioning technique. Unlike these research efforts that cluster low-level paths only, our approach first represents the low-level information into a function call graph. Then applying the coarsening technique to abstract or simplify the function call graph (FCG) into multilevel abstraction graphs. Finally, applying clustering techniques on any coarsen level then projecting the selected cluster back to the original graph (i.e., FCG).

4.2 Background

In this section, we present a brief background on the multi-level graph partitioning technique and define some of the related terms repeatedly used in the rest of this chapter.

Multilevel graph partitions aim to simplify the input graph in order to apply costly partitioning techniques for a smaller problem. The reason behind this is due to limited memory and computational resources which prevented them from applying a set of analysis techniques on the large graph. Several researches have been proven the capability of multilevel graph partitions to quickly produce partition solutions with low cost for numerous complex real-world applications [23, 65, 133]. This technique consists of three phases: coarsening, partitioning, and uncoarsening/refinement. Formally, a multilevel graph works as follows: Consider a graph $G_0 = (V_0, E_0)$. A multilevel graph consists of the following three phases.

Coarsening Phase: The graph G_0 is simplify into a sequence of smaller or coarser graphs $G_1, G_2, ..., G_n$ such that $|V_0| > |V_1| > |V_2| > \cdots > |V_n|$. During this process, vertex matching operations applied to the input graph to create a sequence of coarser graphs. [55].

Partitioning Phase: Once the input graph is aggregated to a suitable size, the initial partitioning algorithm is applied. A partition P_n of the graph $G_n = (V_n, E_n)$ is computed, and V_n partitions into x parts.

Refinement Phase: Once a partition is computed at the coarsest graph, the coarsest graph is projected back to the next finer level graph. The partition for the coarsest graph gives a good starting partition for the next finer level [31]. The partition P_n of G_n

is projected back to G_0 by going through intermediate partitions $P_{n-1}, P_{n-2}, ..., P_1, P_0$.

4.3 Related Work

Analyzing a large amount of the execution paths causes scalability issues due to limited memory and computational resources. Thus, researchers have proposed several reductions and compression techniques to reduce the size of execution paths. Hamou-Lhaji et al. [51] introduced filtering techniques to reduce the number of execution paths. Their approach identifies utilities on the class level that have many direct client classes. Then, filters out the utility components from the execution paths. Similarly, Cornelissen et al. [28] [29] focused on reducing the stack depth of the execution path. While both approaches aim at reducing the execution path size, they may omit essential execution paths. Utility components can play a major role in the overall system implementation by enabling communication between other classes [85]. Chan et al. [20] reduced the size of the execution paths using sampling techniques and provided a tool to sample, visualize, and animate the dynamic traces of the system. Riess [110, 111] on the other hand, focused on reducing the size of the execution paths by encoding the repeated ones. The authors used a comparative approach to find similar paths in the system and encode them together, which resulted in reducing the overall size of the paths. In contrast, our approach applies a coarsening technique to creates multi-level representations of the call graph without altering the main properties of the system structure and its execution paths.

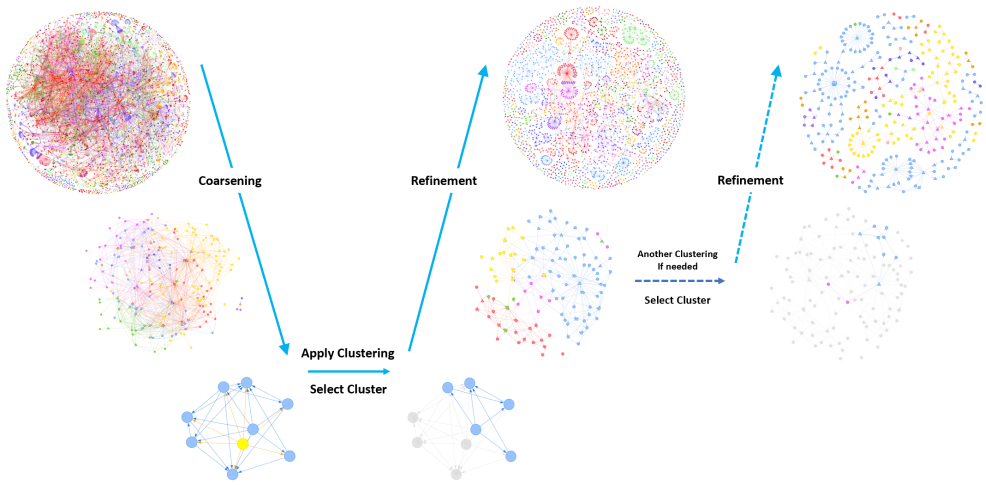


Figure 20: Multi-level Trace Clustering Technique

4.4 Projecting Clustered Paths Approach

For a large system, the developer tends to explore the system in a top-down manner [19, 80]. In particular, the developer first analyzes packages and selects a goal package for further investigation. Then, he/she investigates deeper into the class level and then function level. Our approach follows the same top-down manner for exploring the execution paths. If the size of the function call graph is small and easy to understand, the developer can apply our clustering approach directly to the function call graph. For a more complicated system, the developer can use our coarsening technique to abstract the function call graph. This technique produces multi-level call graphs, such as *Package Call Graph* and *Class Call Graph*. For large systems, the coarsen will go beyond package call graph by using the package hierarchy to coarsen the graph to more than the two levels. After the coarsening has been completed, the user can choose any abstraction level and apply clustering at any level according to their needs, and then these clusters can be

projected to finer levels. Thus, enhancing the user's understanding of the functionality and organization of the overall system.

Before explaining our approach in more detail, Figure 20 illustrates the overview of our approach. Start by coarsening function call graph, applying trace clustering on any coarsen-level, and carried the selected cluster back up to the original function call graph. Both coarsening and clustering techniques are already proposed in Chapter 2 (see Section 2.4.3) and Chapter 3. Thus, In this chapter, we will discuss our Projection/refinement technique.

Our approach constructs different levels of the call graph automatically. First, it constructs a function call graph and then coarsens it into different levels of abstraction (e.x., class and package call graphs). The developer often start from the most abstract level and then selects a specific cluster to investigate it further. Then, a refinement technique is applied to project the clustered graph back to a more detailed call graph (e.g., class or function call graph). To do this, we need to inverse the operation of graph coarsening, known as graph refinement [31]. In graph refinement, nodes expand back into their original representations at the finer-level. We should notice that another clustering and selecting technique can be applied during the projection process. We automate this process by mapping each node in the clustered graph to its corresponding node at the finer-level graph using the attribute namespace. We use the attribute namespace to coarsen call graphs, and therefore, the same attributes are used to expand the node back to its original representation. The implementation details are presented in Algorithm 5.

The algorithm takes two graphs $G_{cluster}$ and G as inputs, assuming that $G_{cluster}$ is

Algorithm 5 Graph Refinement

```
1: procedure GRAPH_REFINEMENT( G_{l(cluster)}, G_{l-1})
2:     R ← DiGraph()
3:     for each edge(u, v) ∈ E_{l-1org} do
4:         i ← u[namespace]
5:         j ← v[namespace]
6:         if i, j ∈ V_{l(cluster)} then
7:             if (i, j) ∈ E_{l(cluster)} or i == j then
8:                 R.add_node(u)
9:                 R.add_node(v)
10:                R.add_edge((u, v))
11:    return R
```

the selected cluster at the coarsest level l, and G_{l-1} is the fine-grained graph of the coarsest graph. For example, if a cluster is selected from the package call graph, the $G_{cluster}$ represents the cluster, while G_{l-1} is the class call graph. The algorithm first constructs a new graph, named R, to obtain the corresponding nodes and edges in both graphs. Each node in both graphs has several attributes (see graph schema in Section 2.4.3.1). For each edge (u, v) of G, we use the attribute namespace as a label for both nodes, $u_{namespace}$ and $v_{namespace}$, and check if their labels exist in $G_{cluster}$. This condition confirms that both nodes u and v are in the scope of $G_{cluster}$. The second condition is to make sure that the corresponding edges are only obtained at each iteration. The same algorithm can be applied to project graph R to the next finer-level. This case is regarded as the function call graph.

4.5 Evaluation

In order to illustrate and evaluate our multi-level clustering approach, we developed a web-based tool, named CodEx (see Chapter 6), and used it to conduct two case studies. We first present the case study Sweethome3D to illustrate that our approach and tool are capable of constructing multi-level abstraction of a call graph and clustering them into hierarchical clusters. Second, we present a quantitative evaluation of two case studies to assess the performance overhead of both single and multi-level clustering execution paths. Although the first case study is a mid-size system, its function call graph considers being complex and harder to visualize and interpret by a developer. Thus, It increases the overhead program comprehension. In previous chapter, we applied the hierarchical clustering process on WEKA function call graph. It was not successfully completed due to limited memory and computational resources. We address this problem in the this evaluation by adapting multi-level graph clustering approach. By applying our approach, we have been able to explore the system's call graphs and alleviate this challenge. Also, facilitate graph clustering analysis.

4.5.1 Case Study: SweetHome3D

In this case study, we illustrate how we alleviate the challenges of understating complex call graphs by adopting a multi-level graph abstraction technique. We applied our approach to SweetHome3D, an interior design application written in Java. The user can design a house in 2D by drawing the plan of the house, adding furniture and home appliance, and then preview it in 3D. It comes with many easy-to-use features, including

import texture and furniture, recording videos, and allowing user-customized preferences, such as setting the system language, fonts, and units of measurement.

Table 12: Call Graphs of SweetHome3D in Package, Class and Function levels

Entity	PCG	CCG	FCG
Nodes	9	196	5,148
Edges	28	901	9,501
Entry Point	2	10	1,517
Exit Point	2	39	2,821

Objective. In this case study, the system includes two versions: desktop and applet. Our goal here is to validate the feasibility of our system in helping the developer to differentiate between the source code of the two different versions, and then further explore the desktop version. This will illustrate that our approach is useful in bridging the cognitive gap and facilitating the comprehension task. Also, we evaluate the efficiency of the proposed abstraction, projection, and visualization techniques in enhancing the overall comprehension of the software system.

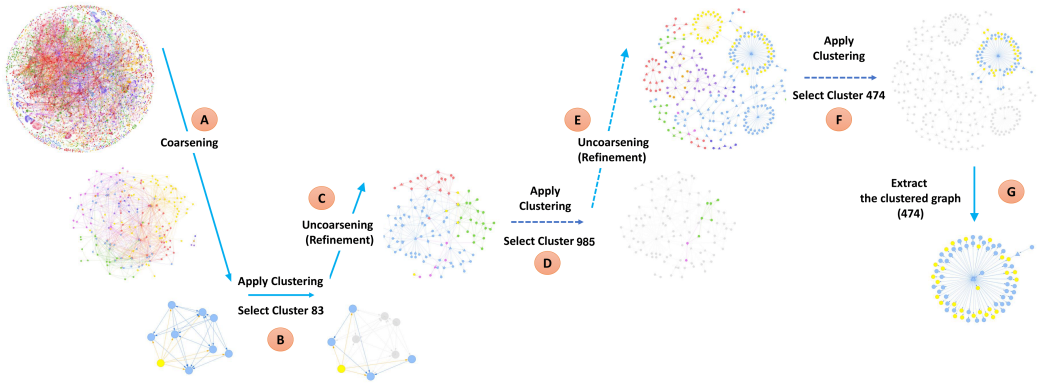


Figure 21: Call Graphs During our Comprehension Process on SweetHome3D

4.5.1.1 Methodology

This case study showcases the task of understanding a system of multiple packages. For a medium to a large-sized system, the developer tends to explore in a top-down manner. The developer starts the program comprehension process by analyzing the packages of the system, then he/she dives deeper and deeper into the classes, functions, and, finally, the source code. Similarly, our approach provides a top-down approach to analyze the system. We start by constructing and analyzing the package call graph, and then particular clusters are selected for more in-depth analysis. The user can navigate from the package graph down to the class graph for further investigation. Finally, we repeat the same process on the class graph to go deeper into a function graph.

4.5.1.2 Results and Discussion

In this section, we analyze the results and report our observations for each level. Figure 21 depicts the workflow of comprehension process on SweetHome3D. (A) Coarsening the function call graph to multi-levels of abstraction. (B) Applying our clustering approach and select Cluster 83. (C) Uncoarsening Cluster 83 to class-level. (D) Applying our approach to the clustered graph and select Cluster 985. (E) Uncoarsening the call graph of Cluster 985 to the function-level. (F) Applying our approach and select Cluster 474. (G) Extract the call graph of Cluster 474.

Package Call Graph. Table 12 shows that the SweetHome3D package call graph has 9 packages (nodes), 28 edges, and 2 entry points that relate to two main methods. The first main method runs the desktop version, and the second main method runs the applet

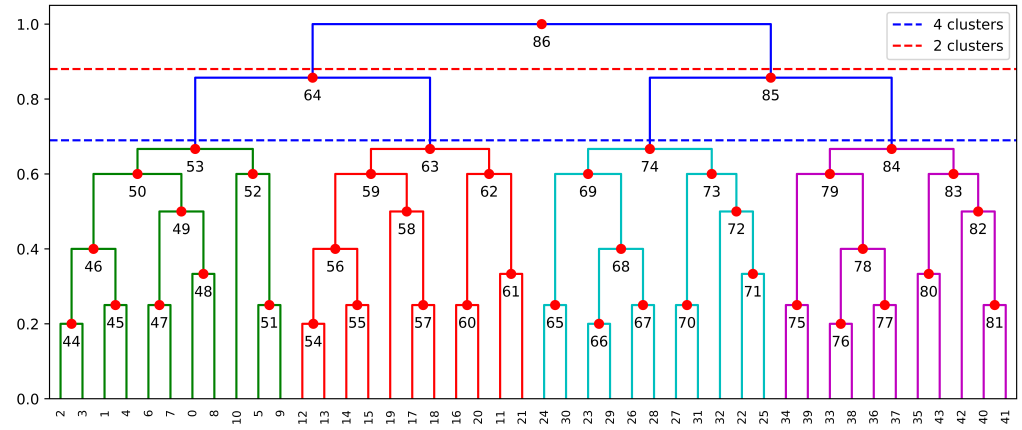
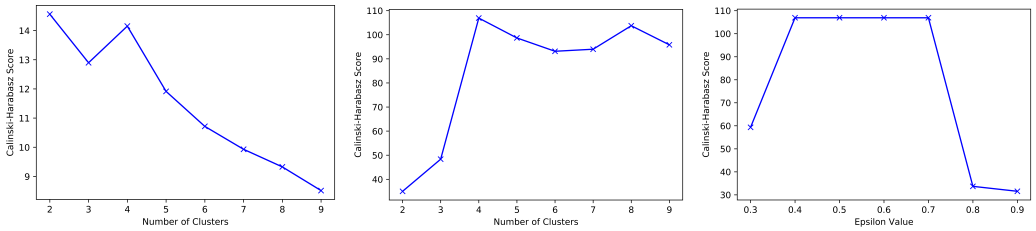


Figure 22: PCG Dendrogram of SweetHome3D



(a) AHC (b) K-Means (c) DBSCAN

Figure 23: Comparing Three Clustering Algorithms on SH3D's PCG execution paths

version. The clustering of the package call graph (PCG) generated from the proposed methods is shown in the form of a dendrogram. The results of the hierarchical clustering are shown in Figure 22. The dendrogram can be divided into two or four groups.

To evaluate our result we use Calinski-Harabasz score for finding an optimal number of clusters. as illustrated in Figure 23. AHC result looks similar to what we suggest earlier. The first-best and the second-best optimal numbers of the clustering are 2 and 4, respectively. However, K-Means pick 4 clusters as its optimal number. The results for DBSCAN look more interesting. The following values of Eps $(0.4, 0.5, 0.6, 0.7)$ suggest 4 clusters as optimal value. We further explored the meaning of the four clusters, namely

Table 13: Structural Characteristics of SH3D's Call Graphs at Different Levels

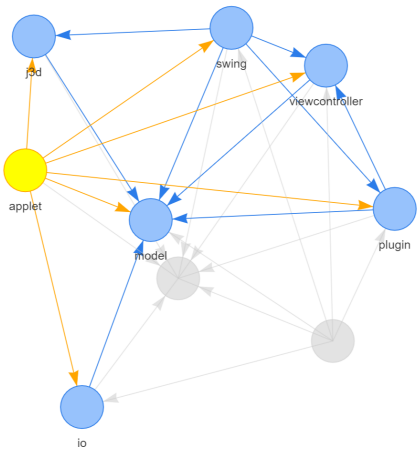
Entity	PCG_{org}	$PCG_{cluster83}$	CCG'	$CCG_{cluster985}$	FCG'	$FCG_{cluster474}$
Nodes	9	5	107	8	301	74
Edges	28	8	247	12	331	73
Entry Point	2	1	3	1	65	1
Exit Point	2	1	30	1	205	71
Paths	44	5	622	10	383	71

Clusters 53, 63, 74, and 84. Figure 24 shows the call graphs of cluster 53,63,74 and 84. The red node represents the package having a main class for the desktop version, while the yellow node represents the package having a main method for the applet version.

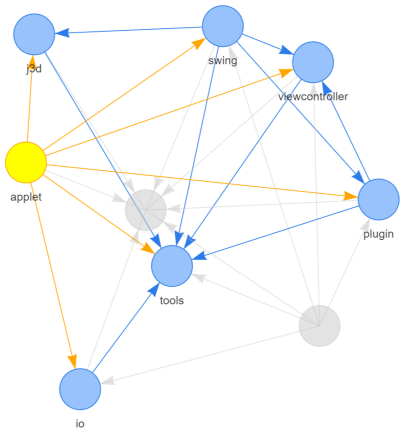
From Figure 24 we observed that our approach successfully differentiates between the two versions of SweetHome3D, desktop, and applet. As seen from the dendrogram in Figure 22, Clusters 53 and 63 were merged into Cluster 64, which represents the applet version. Similarly, Clusters 74 and 84 were merged into Cluster 85, which represents the desktop version.

Class Call Graph. With the call graph abstraction and visualization, users can easily focus on a particular function or part of the system to comprehend the aspects of the system that they are interested in. Thus, The next step is to go deeper into the class level. First, we will choose one of the clusters that belong to the desktop version, Clusters 74 and 84. In this scenario, we will investigate Cluster 84 by selecting one of its sub-clusters (i.e., Cluster 83). Then, convert it to a call graph using the conversion method. After that, uncoarsening/projecting the cluster graph from the package level to the class level using our mapping method, as we discussed in Section (4.4).

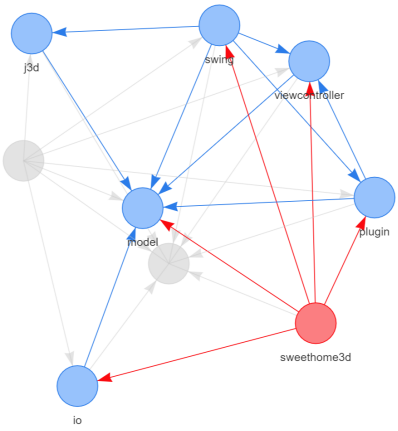
Figure 21.c shows the class graph. It has five different colors which represent the



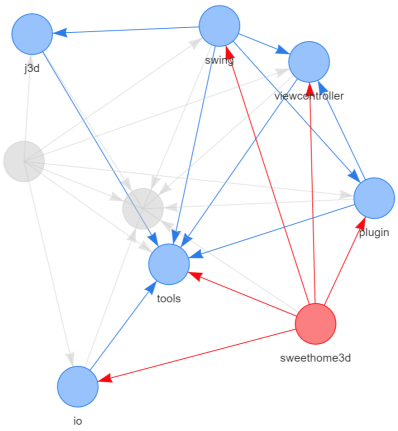
(a) Cluster 53



(b) Cluster 63



(c) Cluster 74



(d) Cluster 84

Figure 24: Call Graphs of Clusters 53,63,74 and 84

following packages: The green nodes are in the 'io' package, the yellow nodes are in the '$tool$' package, the blue nodes are in the '$swing$' package, the red represents the '$j3d$' package, and the pink nodes belong to the '$sweethome3d$' package.

Comparing the uncoarsened class call graph (CCG') to the original class call graph (CCG), the number of nodes was decreased by 54%, from 196 to 107, while about 27% of the edges were decreased, from 901 to 247. In addition, CCG' *is more manageable and more specific to a certain domain. Hence reducing the effort to understand it*. To further investigate CCG', we apply our approach to this call graph. Then, we randomly select a cluster, 985 (see Figure 21.d). This cluster has eight classes from two different packages ($sweethome3d$, and io). The green classes are related to reading and setting the default user preferences in the system.

Function Call Graph. To investigate the function level of the selected cluster, we project ${CCG'}_{985}$ to produce a function call graph (FCG_{985}). Figure 21.e shows the result of uncoarsening the graph. The produced graph has eight different colors that represent classes, to which functions belong. We notice that the graph has three big star networks. We apply our approach to see if this call graph can be partitioned further. Our approach can successfully partition the call graph. For example, Cluster 474 represents one of the main star networks. Using our visualization tool, we found that the cluster handles the user preferences functionality, such as setting the system's language or changing the measurement units. Another interesting observation is that the blue nodes represent the "set" methods that belong to class 'FileUserPreferences,' while the yellow nodes represent the "get" methods that belong to another class named 'DefaultUserPreferences'.

4.5.1.3 Execution Time Overhead

To evaluate and compare the performance overhead of both single and multi-level clustering execution paths, we applied both approaches to SweetHome3D [35], an interior design application written in Java.

We want to compare the performance between single-level clustering and our multi-level clustering methods. To do this, We apply the trace clustering approach in two different methods. The first method is similar to the approaches proposed by us in Chapter 3 as well as these research [38, 42]. These approaches focused on applying trace clustering on the low-level (i.e., function level). The second method is our proposed method which considering clustering different levels of abstraction. To compare the execution time of these two approaches we consider all the processes that we applied in our case study. Beginning with construction the call graph till selecting the cluster of interest.

During the comprehension tasks in the SweetHome3D case study, we measured the time cost for each process that we applied. As shown in Table 14, "Hierarchical Clustering" represents the vast majority of the processing overhead. This is due to the high computation complexity of agglomerative hierarchical clustering.

4.5.1.4 Result and Discussion

In this section, we discuss the evaluation results and our observations. Table 14 shows the execution results of the experiment, This experiment was carried out on a machine with 2.3 GHz quad-core CPU with 16 GB memory. As we expected, Table 14 shows that our approach is significantly faster than the single-level-based technique with

Table 14: Performance Comparison Between SL and ML Method for SweetHome3D

Approach	Single-Level	Multi-Level		
Process/Level	FCG	PCG	CCG'	FCG'
Call Graph	0.291	0.024	0.170	0.291
Execution Paths	146.390	0.002	0.342	0.854
One Hot Encoded	71.613	0.012	0.474	0.340
Hierarchical Clustering	3211.462	2.171	10.412	6.119
Generate Subgraph	0.459	0.036	0.009	0.077
Total	3430.215	2.245	11.407	7.681

an overall performance of 53.52 minutes versus 21 seconds.

The former single-level-based technique, which relied on function level to cluster traces, was not easy to explore or navigate its call graph nor hierarchical view due to the enormous size of nodes, edges, and paths. Our proposed approach, on the other hand, makes exploring the call graph less complex for developers to comprehend. Moreover, less rendering time compared to the former approach.

In the multi-level-based technique, we start by analyzing and clustering execution paths of the most abstract level (i.e., coarsest-level). We observe that the system can be cluster into two large clusters. Each of which represents the different domains. Figure 22 shows the result of the clustering in the dendrogram. Using our tool to map these clusters to the call graph, we found that cluster 64 represents the applet version while cluster 85 represents the desktop version. This observation is almost impossible to identify if we use the single-level-based technique.

By examining the results of SweetHome3D shown in Table 14, we observe that the entire processing time of applying the single-level method is around 57.17 minutes. The

longest sub-process time, 53.52 minutes, representing (93.62%) of the entire processing time, was taken on the hierarchical clustering process.

For the multi-level method, the time cost of the overall process is around 21 seconds. Up to (54%) of all levels belong to $CCG^{'}$. This is due to the large number of execution paths, which led to an increase in the hierarchical clustering process overhead. Nevertheless, overall, the time required to run the tool analysis processes, end-to-end, can be considered efficient comparing to the single-level method.

Our Multi-level method were able to: (1) remove irrelevant information, (2) stay focused on a certain domain, and (3) reduce computation time due to the smaller call graph. This will reduce the cognitive effort to understand the call graph of a software system.

4.5.2 Case Study: WEKA

In this case study, we illustrate how we alleviate the challenges of understating complex and large call graphs by adopting a multi-level graph abstraction technique. We applied our approach to WEKA, which is open-source software that provides a set of tools for data mining tasks including data preprocessing, classification, regression, clustering, association rules, and visualization. It helps to develop machine learning techniques and apply them to real-world data mining problems.

Objective. In the previous chapter we try to apply our clustering technique to WEKA case study. However, we could not complete the clustering process due to memory limitations and resources. In this case study, we want to see if our multi-level approach

Table 15: Structural Characteristics of Call Graphs at Different Levels in WEKA

Entity	$LVL4$	$LVL4_{c2439}$	$LVL5^{'}$	$LVL5_{c1516}$	$LVL6^{'}$	$LVL6_{c8146}$	$LVL7^{'}$
Nodes	32	6	58	14	228	22	348
Edges	134	7	129	21	531	24	453
Entry Point	3	1	14	1	108	1	121
Exit Point	3	1	7	1	42	15	125
Paths	1952	4	936	36	5414	60	2526

can be able to explore such a large system without running out of resources and memory, and with less time.

4.5.2.1 Methodology

Our methodology of this case study is similar to what we did in the SweetHome3D. In particular, We start applying our coarsening technique that constructs seven different levels of abstraction. Since it is easy to understand and it is not much different in the number of nodes and edges between the first three abstract levels, we start analyzing the fourth coarsest level, and then particular clusters are selected for more in-depth analysis. The user can navigate from the coarsest graph down to the next finer graph for further investigation. Finally, we repeat the same process on the finer graph to go deeper into the next finer level till we reach the function graph. As we mention previously, in this case study, we are more interested in measuring the performance of applying our approach to such a large case study. Thus, Table 16 shows the result of analyzing the WEKA system. we measured the time cost for each process that we applied from the fourth coarsest graph till we reach the function call graph.

Table 16: Performance Comparison Between SL and ML Method for WEKA

Approach	Single-Level	Multi-Level			
Process/Level	FCG	$LVL4$	$LVL5'$	$LVL6'$	$LVL7'$
Call Graph	15.996	0.039	0.102	0.087	0.045
Execution Paths	10169.478	0.671	0.363	0.926	0.005
One Hot Encoded	9505.750	0.301	0.255	1.153	0.003
Hierarchical Clustering	-	37.304	14.169	48.001	5.190
Generate Subgraph	-	0.092	0.124	0.043	0.098
Total	3430.215	38.407	15.013	50.21	5.341

4.5.2.2 Result and Discussion

In this section, we discuss the evaluation results and our observations. Table 16 shows the execution results of applying our approach on WEKA system, The experiment was carried out on a machine with a 2.3 GHz quad-core CPU with 16 GB memory.

By examining the results of WEKA system, we could not complete the clustering process using the single-level approach due to limited memory and computational resources. The time cost of construction the function call graph is around 16 seconds. Generating all possible execution paths takes around 2.8 hours while encoding all paths takes 2.6 hours.

In contrast, adopting the multi-level abstraction allows us to explore the system at different levels and alleviate the performance overhead challenges. Table 16 shows that the multi-level-based technique is significantly faster than the single-level-based technique with an overall performance of 1.7 minutes.

4.6 Conclusion

In this chapter, we addresses the need for path reduction techniques in program comprehension. We discuss the limitations of existing solutions and introduce a new reduction technique that enables analyzing the execution paths of a medium or large-scale software system. we conduct two complex large case studies, we illustrate how we alleviate the challenges of understating complex call graph by adopting a multi-level graph clustering technique. Our multi-level approach were able to: (1) remove irrelevant information, (2) stay focused on a certain domain, and (3) reduce computation time. This will reduce the cognitive effort to understand the call graph of a complex and large software system.

CHAPTER 5

TOPIC MODELING FOR CLUSTER ANALYSIS

5.1 Introduction

Clustering the call graph provided multiple abstraction levels that facilitated understanding the software system at several levels of granularity. However, it was evident to us, as developers trying to understand the system, that we still needed to investigate the functions composing each of the clusters in order to understand the overall functionality of the cluster. This requirement was indeed the main motivation to provide a proper labeling technique for the clusters.

Clustering algorithms give no indication or significance of what the clusters are. Thus, in this chapter, we will analyze and interpret the meaning of the resulted clusters. Cluster labeling [72] is the process of generating labels for a cluster of interest, that is, a set of terms that represent a meaningful description to a cluster. It aims to understand the clusters by generating labels that reflect the meaning of each cluster. This chapter proposes an approach to automatically provide labels for clustered execution paths that are presented in previous chapters. It serves as an indicator reflecting the quality as well as the success or failure of the clustering process. We take a first step towards determining the suitability of topic models in analyzing software through clustered execution paths by performing a case study on two systems, SweetHome3D and jMonkeyEngine system, a 3D game development engine written purely in Java. We experiment with two popular

topic models: LDA, BERT, and a combination of both. We found out all topic models indeed generate meaningful topics to describe the functionality of a cluster with a marked improvement for the combined model (BERT+LDA) in respect of coherence score.

5.2 Related Work

A wide variety of topic modeling techniques have been proposed to support varied software engineering activities, including clone detection [68], testing [22], software evolution, software traceability [8] [126], and many others tasks [21]. Researchers have also proposed some approaches to support program comprehension.

Maskeri et al. [84] propose an approach that helps new developers to understand the functionality of large legacy software systems. They use LDA to extract business domain topics from source code. Tian et al. [127] use topic modeling in open source repositories to automatically categorize the software systems. The authors extract topics from several software systems using the LDA model. These topics were grouped based on their word distributions into meaningful categories that represent different types of software applications. Savage et al. [117] introduced $Topic^{XP}$, a visualizing tool for topic modeling. It helps developers understand the system by extracting and analyzing unstructured information (i.e. identifiers and comments) from the source and using the LDA model to automatically generate topics. Wang T, Liu Y [135] introduced JSEA, Java Software Engineers Assistant. The tool uses LDA-based topic modeling to mine commented source code to construct a project overview with search capability.

Other interesting works exist in this area, including semantic analysis on execution

paths. Kuhn et al. [71] proposed an approach that analyzes the execution paths using an information retrieval technique, namely Latent Semantic Indexing (LSI) [32]. Their approach considers only method names to represents the execution paths as documents. They apply LSI on these documents and uses a hierarchical clustering algorithm to cluster them. To visualize the similarity between paths, they use a grayscale heatmap. The darker the color, the more similar these two documents.

Another work that is very similar to our approach is the Sage tool [38]. Sage is a dynamic analysis approach that focuses on building hierarchical abstractions from function calls. Their approach labels the hierarchical abstraction level (i.e., cluster) with a proper functionality name based on the frequency of the method names in the cluster. To provide the most relevant and distinguishing terms for a cluster, they adopt TF-IDF (term frequency-inverse document frequency) [141] metric and extract the top 20 terms as the final label set for the cluster.

Our approach differs from previous approaches in two ways. First, We focus on clustering static execution paths. Second, Our labeling cluster considers not only the method names to label a cluster but also the comments and the identifiers inside each method that is part of the cluster.

5.3 Proposed Approach

We have already discussed how a call graph of a system can be clustered into a multi-level of granularity. Chapter 3 presented clustering algorithms that have been implemented in CodEx. The algorithm accept a call graph as input and create a clustered

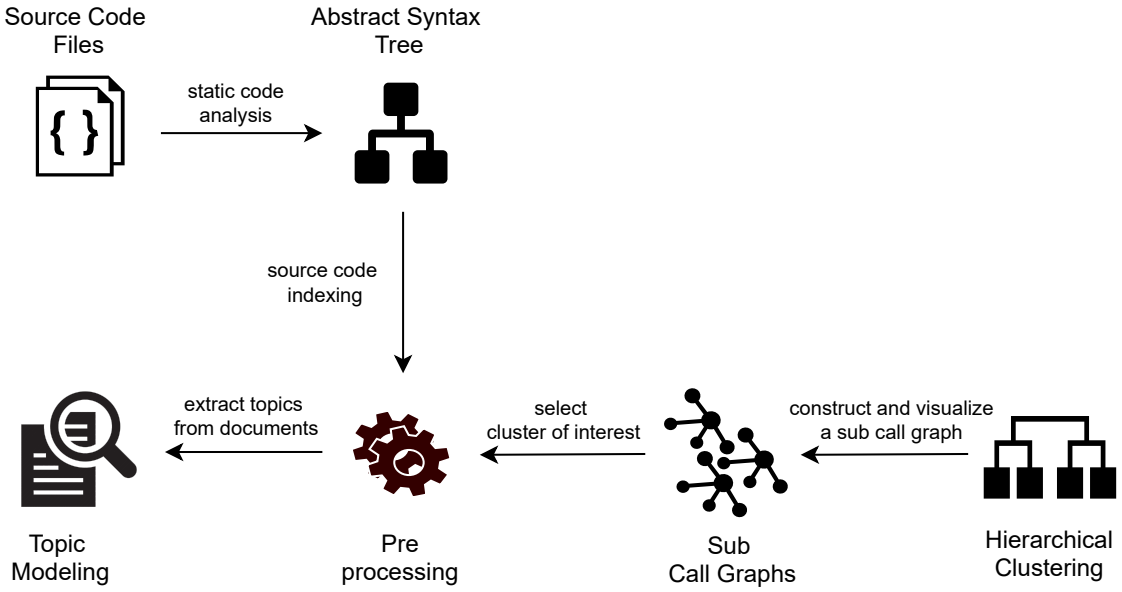


Figure 25: High-level of Topic Modeling Approach Workflow

call graphs as output. Since our goal focuses on understanding the overall functionality of the cluster, Figure 25 shows the portion of our overall approach that is related to the topic modeling part. We first need to select a cluster of interest. Then, we use the unstructured information (i.e., identifiers and comments) from each method belong to the selected cluster to help determine the functionality of a cluster. This can be done by tokenizing the unstructured information and applying preprocessing steps described in 5.4. Then, feed it into a topic model to generates topics. In the following sections, we discuss more details of our approach.

5.4 Source Code Indexing

Before applying topic modeling to the clusters, we need to perform several pre-processing operations. These operations are necessary to improve the quality of the topic

```
//Sets up the attributes for the class
   public AddHCPAction(DAOFactory factory, long loggedInMID) {
      this.personnelDAO = factory.getPersonnelDAO();
      this.authDAO = factory.getAuthDAO();
   }
```

```
set up attribut
add hcp action dao factori factori log mid
personnel dao factori get personnel dao
auth dao factori get auth dao
```

Figure 26: Source Code Indexing [81]

modeling result. Natural language processing (NLP) techniques are usually used to clean and remove noise in the source code. Thus, we apply three preprocessing steps described in the following subsections. The result of this preprocessing is a file where each row contains the fully qualified name of a method and its tokens. Late, we will use this file to map each method to the path to which they belong.

5.4.1 Removing Programming Language Keywords

In addition to the syntax information provided in the source code, it also contains unstructured data such as natural language identifiers and comments [86]. These identifiers and comments have meaningful information about the code and can reflect the semantics intention of the code [144]. Thus, we isolate identifiers and comments by removing syntax and keywords of programming language (e.g., public, static, for, if, and int). We also use NLTK toolkit [13] to remove common English language stop words (e.g., the, it, on, can, set, get) to reduce noise.

5.4.2 Word Splitting

In this step, we tokenize each word based on common naming practices. To split multi-word identifiers, we use one of the common ways to split identifiers by following programming language naming conventions [33]. In particular, Identifier names are split into multiple parts based on common naming conventions, including dot separators (swing.SettingView), $CamelCase$ (SettingView) in Java programming language.

5.4.3 Stemming Identifier

In information retrieval (IR), stemming is the process of stripping affixes (i.e., prefixes and suffixes) from words to form a stem [138]. In other words, It reduces the words to their root form. For example "retrieval", "retrieved", "retrieves" reduce to the stem "retrieve". Stemming plays an important part in data pre-processing. It reduces noise in your data by reducing inflectional forms and related forms to the common base of a word. Thus, it can improve the quality of the topic modeling model. We will be leveraging Gensim [108, 123] package, to perform stemming on the English text. Gensim is a open source Python library for unsupervised topic modeling. The library also provides implementations for various text prepossessing algorithms for cleaning the data.

5.5 Topic Modeling on a Cluster of Execution Paths

5.5.1 Latent Dirichlet allocation

In order to apply LDA to a cluster of execution paths, we represent a cluster as a collection of documents (i.e., execution paths). To understand the clustered execution

paths, we need to investigate their low-level implementations (i.e, function level). Therefore, when we select a cluster of interest. Our approach checks the level of granularity and sees whether this cluster was selected from coarsening level or not. If yes, then we need to project each path in the cluster back to the function level using our projection technique described in Chapter 4. After that, we retrieve the tokens of each method in the path. Specifically, our inputs can be defined as the following:

- A corpus C is a set of documents (i.e execution paths) denoted by $C = d_1, d_2, ..., d_n$
- A document d, which corresponds to a execution path, is a sequence of words denoted by $d = f_1w_1, f_1w_2, ..., f_nw_n$ where w_i is the ith word that is part of function f_i. These words defined to be an identifier or a word from a comment.

$$C = \begin{pmatrix} f_1w_1 & \cdots & f_2w_1 & \cdots & f_nw_n \\ \vdots & \cdots & \vdots & \cdots & \vdots \\ f_mw_m & \cdots & f_mw_m & \cdots & f_mw_m \end{pmatrix}$$

For the implementation of the LDA model in this experimental analysis, we used gensim [107] package. Gensim is a open source Python library for unsupervised topic modeling. The input of Gensim is a corpus of plain text documents. The library also provides implementations for various algorithms, including neural word representations (word2vec [88], fastText [16]), as well as latent semantic analysis (LSA), latent Dirichlet allocation (LDA), TF-IDF, and Random Projections [78] to discover semantic topics of documents.

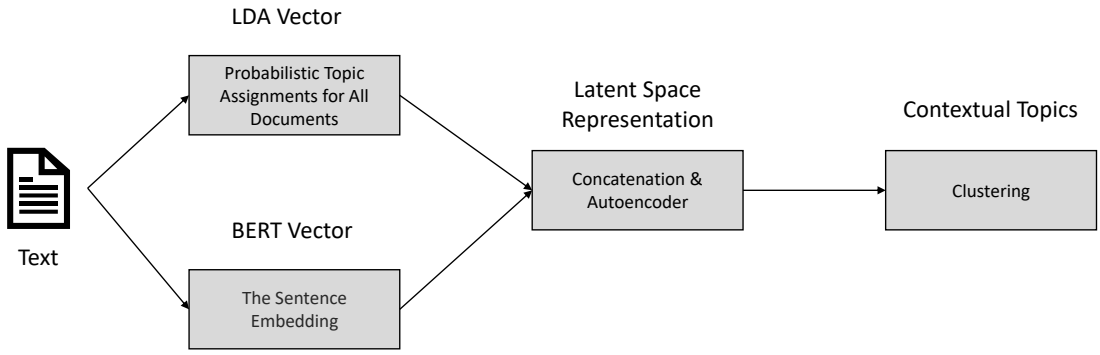


Figure 27: Contextual Topic Model

5.5.2 Contextual Embeddings

LDA users Bag of Words (BoW) document representations as model input. This type of model based on the co-occurrence of words, and yet, it does not consider the sequence of words. This drawback makes LDA fail to learn contextual information in sentences. Thus harder to interpret. In contrast, contextualized language model such as BERT (Bidirectional Encoder Representations from Transformers) uses word embeddings techniques that consider not only the word but also its context. Adding contextual knowledge to the model can improve coherence of the generated topics.

To do this, we first generate the probabilistic topic assignments for all documents from the LDA model. Second, we used Pre-trained BERT. It is widely used for many Natural Language Processing (NLP) tasks. We use the 'sentence-transformers' [109] package for document-level embedding. We embed our execution paths into vector space. Third, we concatenated both vectors. The concatenation process generates a higher dimensional space, where information is sparse and correlated, Thus, we used an autoencoder to learn

a lower-dimensional latent space representation of the concatenated vector. Fourth, we cluster them to identify their similarity structure in the vector space. Finally, generate the topics within these clusters.

5.6 Case Study: Sweethome3D

We conducted a case study on the SweetHome3D system to see if our approach can reflect the meaning of the selected cluster. Also we want to show how the automatic labeling and multi-level visualization of clustered call graph are complementary to comprehend the cluster of a system.

5.6.1 Selecting Cluster of Interest

To be able of selecting a cluster of execution paths, we first used our tool, CodEx. The tool can automatically analyze the source code of the system, construct call graph of the system. Then, extracting and clustering its execution paths. After clustering the system we randomly choice one of clusters and converted to a call graph using CodEx. The selected cluster is made up from 54 execution paths. In particular, the clustered graph consists of 29 nodes/functions and 34 edges. Figure 28 depicts the selected cluster in term of call graph.

5.6.2 Results and Discussion

In this section, we report the main observations for the different experiments that we performed. We show that adding contextual information to LDA model provides a significant increase in topic coherence score.

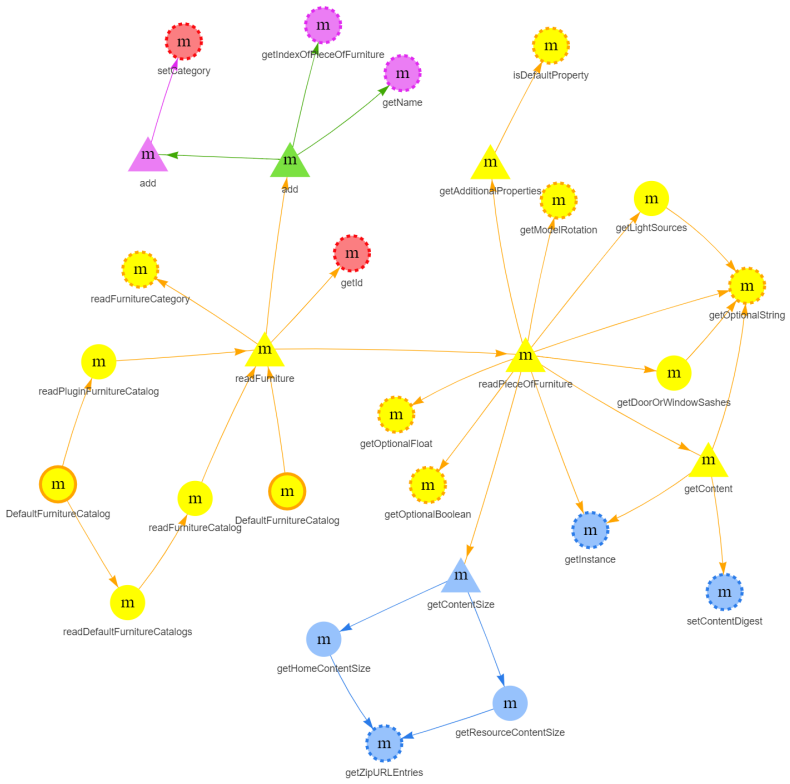


Figure 28: Clustered Graph Responsible of Reading Furniture Component

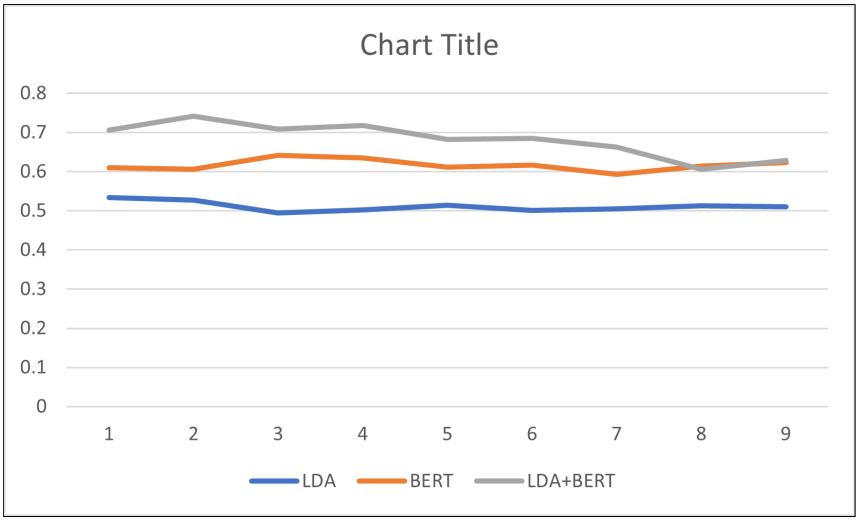


Figure 29: Coherence Score of each Model Across Different Number of Topics

We compute the topic coherence for different number of topics and choose the model that gives the highest topic coherence score. Figure 29 presents the scores of different values of K topics with a range from 1 to 9. The highest score refers to the optimal value of K. Based on the coherence score, the K value of 1 yielded the highest score for LDA model, while *k=3, 2* yielded the highest score for BERT, and LDE+BERT, respectively.

Table 17: Extraction of Top Ten Keywords with LDA, BERT and LDA+BERT

Model	Topic	Keywords
LDA	T1	furniture resource catalog read bundle piece exception key miss index
BERT	T1	resource furniture bundle catalog read exception piece key miss parse url
	T2	resource furniture catalog read bundle piece key default index identify add
	T3	resource furniture url size content read catalog exception file entry
LDA+BERT	T1	resource furniture bundle catalog read exception piece key miss parse
	T2	resource resource furniture catalog read bundle piece key index exception

5.7 Case Study: jMonkeyEngine

In this section we present a case study to show how our labeling and multi-level abstraction of call graph techniques simplified the task of understanding the subsystem structure and functionality of the complex call graph of a program. We conducted a case study on the jMonkeyEngine system [106]. It is a 3D game development engine written purely in Java. jMonkeyEngine allows you to develop 3D games on different platform including desktop, web, and mobile. It consists of 1,705 java file, and has 201,324 LOC and 122,156 comments.

5.7.1 Selecting Cluster of Interest

To be able of selecting a cluster of execution paths, we need first to analyze the source code of the system, construct call graph of the system. Then, extracting and clustering its execution paths. This can be done automatically using CodEx. More details were described in previous chapters of this dissertation. After clustering the system we randomly choice one of low-level clusters and converted to a call graph using CodEx. The selected cluster is made up from 835 execution paths. In particular, the clustered graph consists of 655 nodes/functions and 773 edges. Without a proper label for this cluster, developers need to investigate the low-level functions at the source code in order to figure out the cluster's overall functionality.

5.7.2 Data Preparation for Topic Models

In this step we first extract all execution paths of the selected cluster. Second, To convert an execution path into a document, we extract identifies and comments of each method belong to the path. Third, transform the textual data into a proper format for the topic models to consume. This can be done by tokenizing the text and applying preprocessing steps described in the subsection 5.4. The outcome of this process is 835 documents with 915 unique tokens. Finally, we apply LDA, BERT and a combination of both to understand the functionality of this cluster.

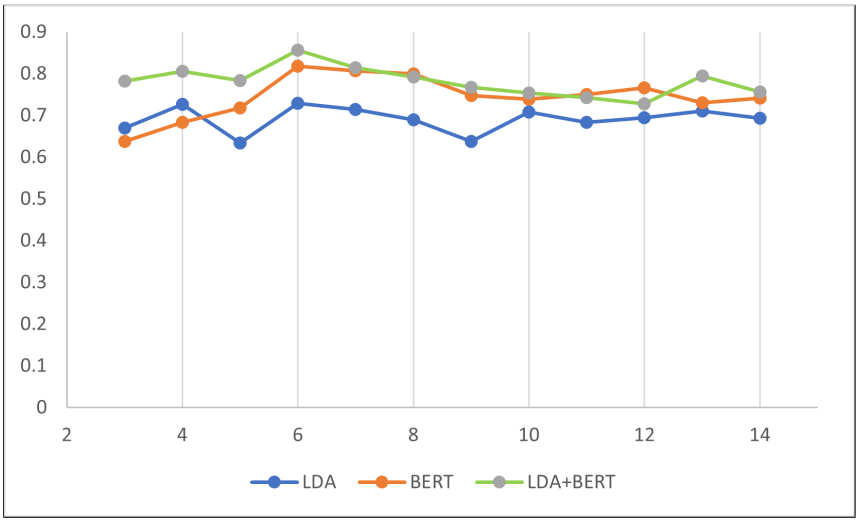


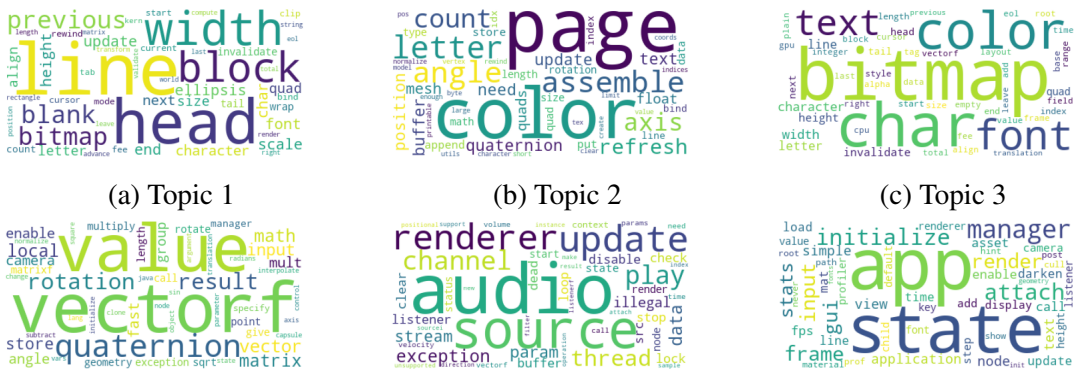
Figure 30: Coherence Score Across Different Number of Topics

5.7.3 Results and Discussion

In this section, we report the main observations for the different experiments that we performed. As we expected, adding contextual information to LDA model provides a significant increase in topic coherence score.

In order to determine the number of topics, we compute the topic coherence for different numbers of K topics. Figure 30 presents the scores of different values of K topics with a range from 3 to 14. The highest score refers to the optimal value of K. Based on the coherence score, the K value of 6 yielded the highest score for all topic models, while k=*5,3,7* yielded the lowest score for LDA, BERT, and LDE+BERT, respectively. Thus, we chose 6 as the number of topics.

We use WordClouds [92] to represent the generated terms in each topic. This visualization can assist us to explore textual analysis by identifying words that frequently appear in each topic. In particular, The more frequently a word appears in the topic, the



(a) Topic 1 (b) Topic 2 (c) Topic 3

(d) Topic 4 (e) Topic 5 (f) Topic 6

Figure 31: Generating Six Topics Using LDA

bigger it will become. As we can see in Figure 31, We generate 6 topics for the cluster. Figure 31.a,b and c, shows the top words in topics 1,2 and 3. It appears they contain tokens about text characteristics and styling. Figure 31.d shows the result of topic 4. It contains tokens such as vector, matrix, and value. Figure 31.e illustrate topic 5 which it seems is about rendering audio. Finally, Figure 31.f shows some words about initializing app state.

To verify the labels and topics we project the clustered call graph to the coarsen level. Figure 32 shows the projection result. The class level consists of 49 nodes and 89 edges. It has nine different colors which represent the following packages: The green nodes are in the $'audio.openal'$ sub-package, the red nodes are in $'audio'$ package, the blue nodes are in the $'app'$ package, the yellow represents the $'app.state'$ sub-package, the purple nodes belong to the $'font'$ package, the dark blue nodes are in the $'scene'$, while the pink nodes belong to a sub-package of $'scene.control'$. The orange nodes represent the $'math'$, and the magenta nodes are in the $'export'$ package. Comparing the topics

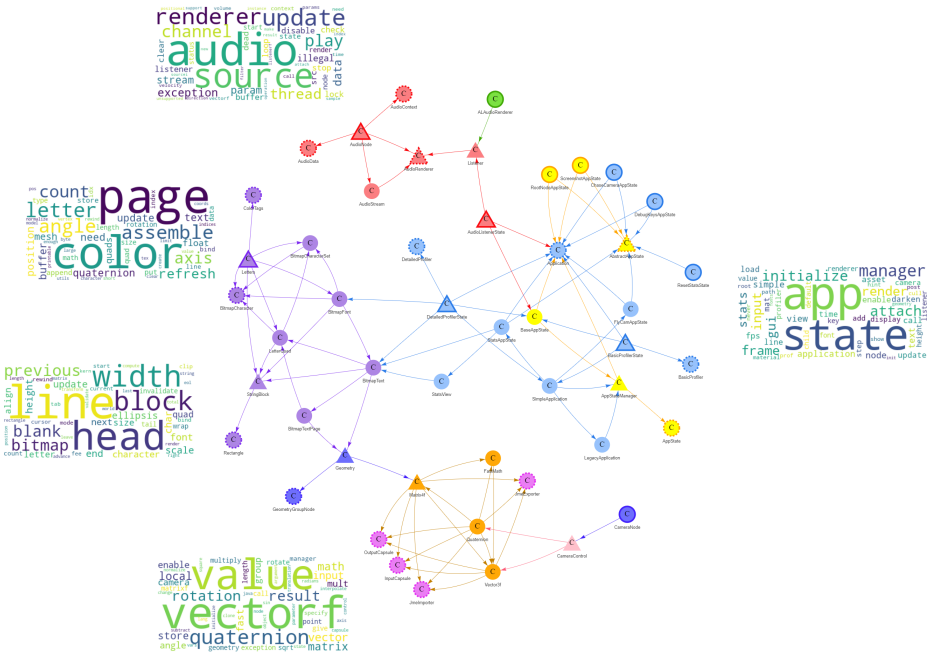


Figure 32: Class Level of Cluster Call Graph

result and the coarsen call graph, we notice that the class level of the clustered function call graph is reasonably close to generated topics.

We first turn our attention to the first three topics, We can easily map words with the class call graph since these words are also used as the class name. In particular, words like $block$, $font$ and $bitmap$ are also showed in classes names such as $BitmapFont$, $StringBlock$ and $BitmapTextPage$.

Topic 4 deals with various aspects of mathematics such as $vector$, $matrix$, $quaternion$ and $rotation$. Looking back to the class call graph, we notice that these terms have similar or closely related to the name of classes in dark blue and orange nodes. These nodes belong to $scene$ and $math$ packages.

In Topic 5, We see both words $audio$ and $render$ are large which means they most

frequently appear in the topic. We notice that these terms have been used as the name of red and green classes. Similarly, in topic 6, we see large terms, namely $'state'$ and $'app'$. These terms are used together as the name of yellow and blue nodes/classes.

We observe that none of the topics shows any words related to export functionally. Specifically, the purple nodes/classes belong to $export$ package, namely $InputCapsule$, $OutputCapsule$, $JmeImporter$, and $JmeExporter$. With more investigation with the function level of the cluster call graph, we found that only 4 out of 655 functions belong to the $export$ package. Moreover, after investigating the code level of these functions, we found out these functions have a short comment or none with a short body and mainly about reading and writing data.

Table 18: Top Ten Keywords with LDA

Topic	Keywords
T1	app state manager initialize attach input render frame gui stats
T2	line head width block previous blank bitmap ellipsis update char
T3	audio source update renderer channel play thread exception data param
T4	vectorf value quaternion rotation result vector matrix math local input
T5	page color letter angle assemble count axis refresh quaternion update
T6	bitmap char color font text character width letter line invalidate

Table 19: Top Ten Keywords with BERT

Topic	Keywords
T1	text line color font width letter quad character char bitmap
T2	state app manager attach input initialize call node line enable
T3	audio source update renderer channel param play thread data exception
T4	quaternion rotation angle math value matrix vectorf result axis fast
T5	line head width bitmap char letter invalidate font character quad
T6	app state enable render manager initialize input frame camera vectorf

Table 20: Top Ten Keywords with LDA+BERT

Topic	Keywords
T1	color quad letter page assemble text count update position quaternion
T2	state app manager initialize attach input render text frame gui
T3	quaternion matrix vectorf rotation value math result vector angle local
T4	line head width block bitmap previous blank char ellipsis font
T5	audio source update renderer param channel play thread data exception
T6	color char bitmap font text character width letter line invalidate

CHAPTER 6

VISUAL EXPLORATION OF SOFTWARE CLUSTERED TRACES

6.1 Introduction

In recent years, a number of trace clustering techniques for program understanding have been published [38, 41, 42, 143]. However, little attention has been given to the area of visually examining and interpreting these clusters in terms of the static dependency graph. Most trace clustering approaches visualize clustering results using flat lists or static hierarchical dendrogram [41]. Which makes it difficult for users to explore, interpret, and navigate. Our work on visual clustering goes one step further and argues that it is not sufficient to just clustering a set of execution paths without facility the process of exploration for the user. Visualization is an effective way to ease the interaction process with source code and hence support comprehension tasks. Our premise is that multi-level graphs clustering and visualizing clusters with the overall system via call graph supports maintenance activities by reducing comprehension time. We developed a fully automated visualization tool that supports clustering-based visual exploration tasks for the execution paths of a system. The tool allows developers to explore, coarsen, and project the clustered paths in an interactive call graph representation. Although in this dissertation, we focus on the Java language. However, our approach can be applied to other programming languages.

6.2 Related Work

There are many researches in the software visualization area that depict the structure of system [17] [89] [119]. Although our visualization tool is mainly developed for clustering-based visual exploration of execution paths, it can visualize the system structure at different levels of abstraction. However, little attention has been given to the area of visually examining and interpreting these clusters. In this chapter, we focus on the most related work to ours that can be categorized into visual clustering in the software engineering domain.

SDVisu [105] is a tool for clustering-based visual exploration of dependency graph whose nodes are files and edges are static dependencies between the files. The tool uses LinLog energy model [97] to cluster the software files. Slob, Govert-Jan, et al [121] propose exploration tool called Narrator. It takes requirements in the form of user stories and translates them into an interactive directed graph diagram using Natural Language Processing (NLP). Another works by Reddivari, et al [104] [103] use visual clustering to explore how landmarks can be identified via static dependencies.

6.3 CodEx: A Visual Exploration Clustering-based Tool

The visualization tool that we described in Chapter 2 has been extended and integrated with our clustering approach described in Chapter 3. We named it CodEx. It consists of three main linked views:

6.3.1 Graph View

CodEx visualizes the call graph of a system at the different levels of abstraction. The tool will render the most abstract level first to avoid the complexity of the software system when first loading its call graphs. Then, the developer will have the choice to render a finer-level graph in a new tab. This will give the user the ability to explore and apply analysis tasks on one level while waiting for the other graph to render on the other tab. Moreover, CodEx provides a set of navigation features that allow the user

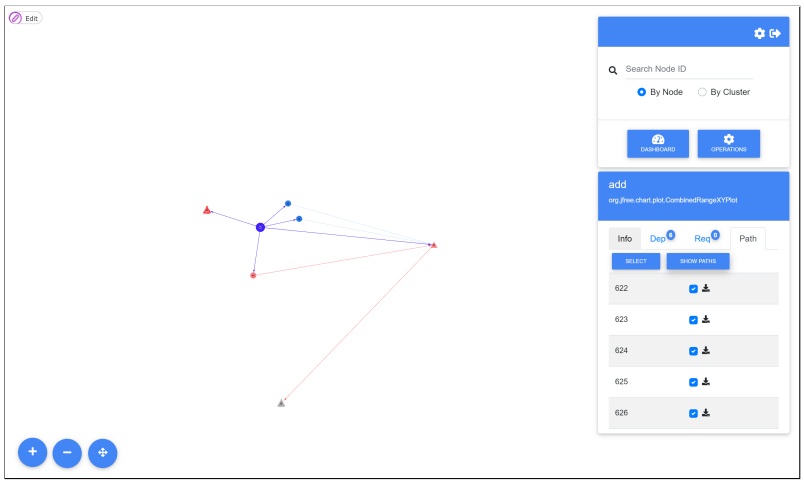


Figure 33: Path View

to interactively explore the call graph including zoom in, zoom out and fit the graph on the screen. When the user clicks on a node, a panel on the right side of the screen will show up. It shows details including type, in-degree, out-degree, and graph matrices. Simultaneously, the tool also highlights the selected node and its neighbors while coloring the rest of the graph with a transparent gray color. It also utilizes the shape and the color of the node to present different proprieties. More details described in Chapter 2.

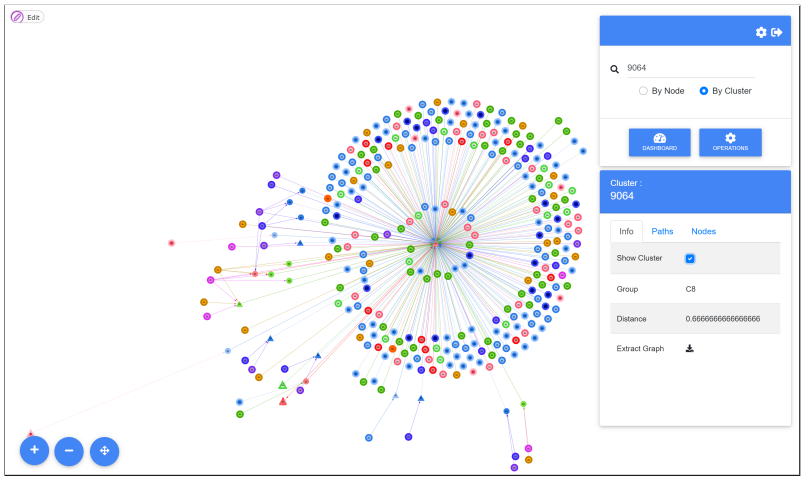


Figure 34: Cluster View

Although node-link representation is commonly used to visualize the call graph of a system, it tends to become highly cluttered when large numbers of nodes and edges are visualized. To address this problem, we provide a set of features to filter irrelevant nodes and edges which allow the amount of information in the graph manageable and easy to explore. Moreover, we have made several enhancements to further facilitate the process of exploring execution paths and clusters in the call graph.

To support developers in navigating through traces and identifying path of interest, CodEx allows the developer to view all or certain paths that belong to a selected node in the graph. In particle, the tool allows hiding all edges and nodes except those that belong to selected paths. Figure 33 shows a side panel of the selected node with a list of all paths that contain the node. The developer can check a path or a set of paths that he wants to view in the graph. Similarly, with cluster view in the Figure 34. After entering the cluster number in the search box, the developer can check the checkbox in the side panel to view only the nodes and edges that belong to the cluster and hide the others.

6.3.2 Clustering Process View

To make clustering the call graph into a multi-level of granularity easy for the user, we integrated the pipeline of the clustering process into graph view. Figure 35 shows the modal window listing all the main clustering processes. To view this window the developer needs to click on the operation button in the top right panel. Each process has states, action, and execution time. The true state tells the developer that the process is already executed and the output of this process is ready to download. If the state is false, then the developer needs to click on the gear icon to execute the process. The estimate column shows the time taken by the process to complete. To view the clustering results in dendrogram, the user needs to click on the eye icon, next to the hierarchical clustering process.

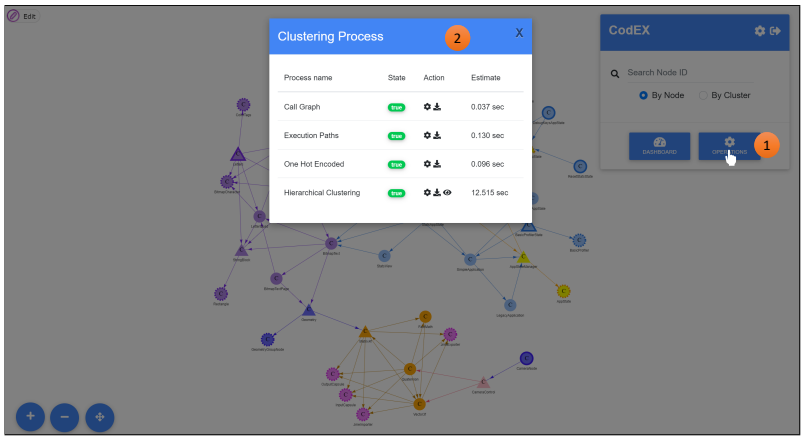


Figure 35: Clustering Process View

6.3.3 Hierarchical View

The dendrogram is one of the popular formats for presenting hierarchical clusters. However, when the size of the graph increases, it becomes difficult to read and identify clusters, especially when horizontal lines connecting clusters overlap [115]. To overcome this issue, we extend our visualization tool to view the hierarchical clustering results. Figure 36 depicts the hierarchical view of the resulted clusters. The top right panel allows the developer to customize the tree view of the cluster such as the distance between the nodes and the height between the levels. The bottom right panel views information about the selected cluster such as the distance and the list of paths or functions that belong to the selected cluster.

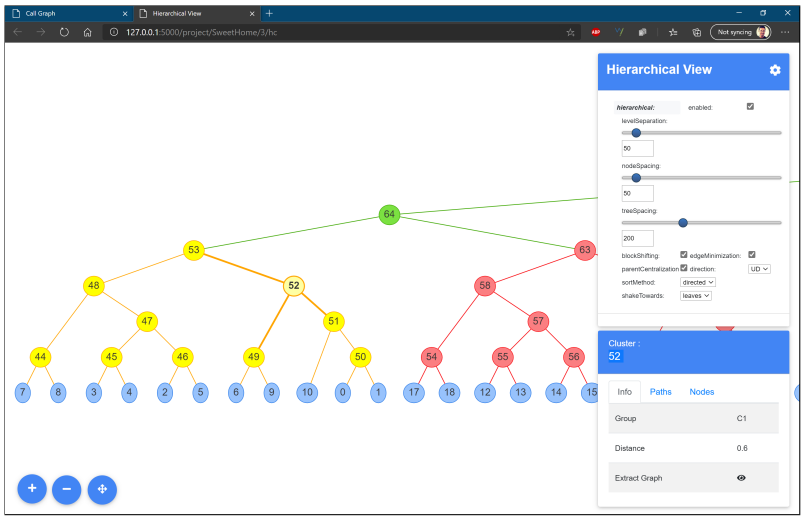


Figure 36: Hierarchical View

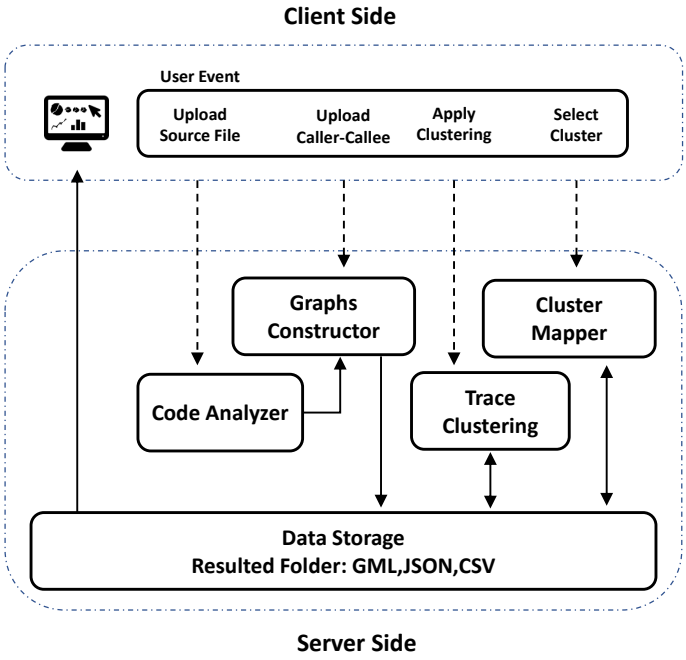


Figure 37: High Level Architecture of CodEx

6.4 Design and Implementation

To implement our visual clustering-based tool, we integrate our clustering approach described in Chapter 3 and 4 with our call graph visualization tool. Figure 37 shows the high-level architecture of our tool. As we can see the tool consists of five components.

Graph Analyzer This component is responsible for reading and analyzing the source code of a system. Then, constructing the caller-callee list. Finally, passing the caller-callee list to the Graph Contractor component. The Analyzer component has only Java Analyzer [44]. However, it can be easily extended to support more programming languages.

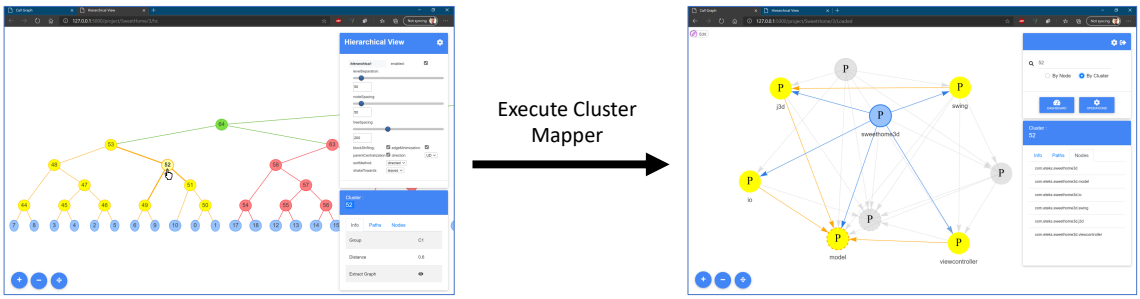


Figure 38: Mapping Clusters to a Call Graph

Graph Constructor This component consists of two main functionality: (1) parsing the edges list to construct a function call graph, then (2) aggregate the function call graph into multi-levels of abstraction such as class-level and package level. While parsing the caller-callee list, we also extract nodes' properties and their relationships using a generic graph data structure, NetworkX [48]. This library was used to manipulate the structure of the call graph. Then, we created a JSON schema to obtain node properties. After constructing the call graph, the results are saved in different file formats, including GML and JSON. We use JSON schema for visualization of the call graph, and GML for manipulate and projection the call graph.

Trace Clustering provides several functionally including collecting execution paths, preprocess, encoded and clustering the data into multi-level of granularity using Agglomerative Hierarchical Clustering algorithm. We implemented with *Numpy* [52, 98] and *Scipy* [130, 131] libraries. The details of implementation of this component was descried in this paper [41] as well as in the Chapter 3.

Cluster Mapper This component is responsible for linking the Hierarchical view with the Graph view. Figure 38 depicts the process of mapping between the views. When

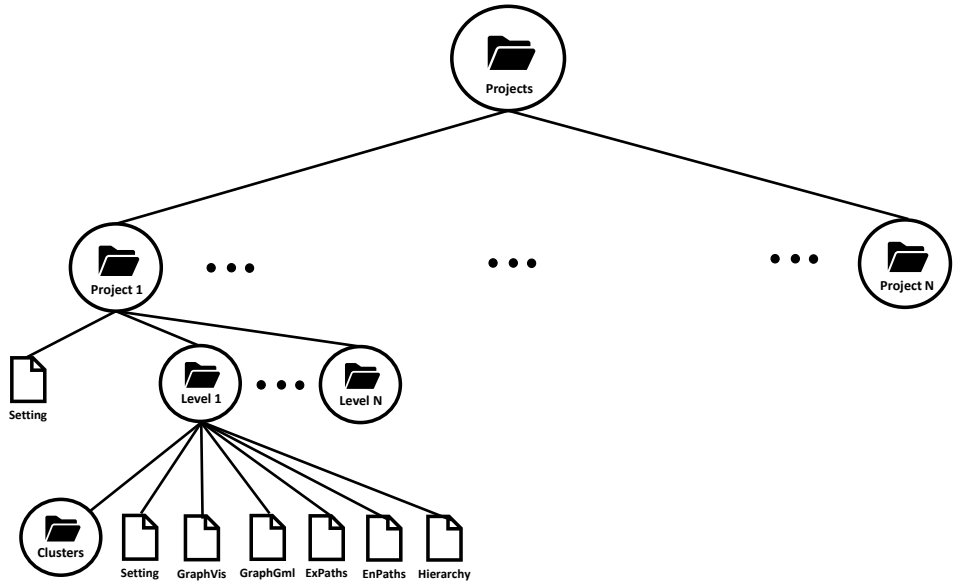


Figure 39: File-based Storage System

the cluster/node 52 is clicked, an event with cluster-id is sent to the server. The Cluster Mapper component will construct a clustered graph by obtaining all the paths that belong to the selected cluster. Then, pass it to the NetworkX graph object to construct the clustered graph. Finally, it saves the clustered graph in JSON format for visualization. Later, when the developer inputs the cluster id in the search box in graph view, the tool will read the file of the cluster and maps it to the call graph of a system.

Data Storage This component is responsible for storing all the data generated from the above components. Our visualization tool user file storage system, also known as file-level or file-based storage format, to store and organize the data. When the user inputs the name of the project and uploads the source file of the project that he wants to analyze, our system will automatically create a folder for that project. Also, it will

create a subfolder for each level. Each level will have its own data such as graph and execution paths data etc. Figure 39 illustrates the structure of CodEx storage system. As shown in the figure, Project 1 has a setting file and a list of subfolders. The setting file has configured metadata related to the current project such as the path of the project to facilitate accessing the project files by telling the computer exactly where the files of the current project are kept.

6.5 User Study

A basic objective of carrying out user studies is to seek insight into how a specific technique is useful [69]. Similarly, our user study aims to gain insight into how useful our approach and tool are based on user experience feedback. Moreover, It can help us assess the strengths and weaknesses of our visualization tool and can guide future efforts to improve existing techniques.

In this section, we evaluate the usefulness and usability of our approach and tool using a user study. The user study involved 18 participants who carried out a set of tasks and then answered a questionnaire. The questionnaire included 12 questions to assess the usefulness of the tool and 10 questions to assess its usability using the System Usability Scale (SUS) [18].

6.5.1 Participants

The user study was conducted by inviting real software engineers with at least three years of experience to test our tool and then answer the survey. We reached out to more than 25 software engineers. Only 18 opted to participate in the study. 13 out of 18

participants mentioned their industries. The participants belong to more than 11 international industries worldwide, including Google, Apple, and others. However, we have not received sufficient responses from our participants regarding their age and gender, and thus we did not want to include incomplete statistics in the study. Our participants were experienced (39% more than 5 years; 61% 3 to 5 years).

6.5.2 Procedure

To make the tool accessible to all participants, we deployed it on a Linux virtual machine (VM) running Ubuntu 18.04 LTS on Azure.

Before the participants started the study, they had to complete several small tasks. First, they had to sign a consent form. Second, they were asked to answer some questions to gather demographic information, such as work and level of experience, and assess their views on the research area. Third, the participants were asked to watch a short video clip that provides a brief overview of our visualization tool.

To evaluate the tool in this study, participants were provided with a set of software comprehension tasks and a questionnaire. The tasks were intended to collect data about the usefulness and the usability of the tool. The given tasks are listed in Table 22. The subject study was the SweetHome3D application (refer to Subsection 5.2 for more information on this system). The participants then completed the questionnaire while using the tool. The responses of the participants were collected using Google Forms.

Table 21: Relation Between Tasks and the Activities Defined by Pacione et al.

ID	Activity Description	Compatible tasks (IDs)
1	Investigating the functionality of the system	T5
2	Adding to or changing the system's functionality	T4
3	Investigating the internal structure of an artifact	T3
4	Investigating dependencies between artifacts	T1,T2
5	Investigating runtime interactions in the system	-
6	Investigating how much an artifact is used	T1,T2,T4
7	Investigating patterns in the system's execution	-
8	Assessing the quality of the system's design	T1,T2
9	Understanding the domain of the system	T5

6.5.3 Tasks

Each participant was asked to perform a set of tasks using our tool. When we designed our tasks, we kept two main goals at the core of the study: 1) The tasks should be representative software comprehension tasks, and 2) they should exercise all of the tool's features. To achieve the first goal, We designed our comprehension tasks based on a common comprehension framework from Pacione et al. [99]. They studied several sets of tasks used in comprehension evaluation literature and software visualization. Table 21 shows that our tasks cover most comprehension activities in Pacione's framework. The uncovered activities are mainly concerned with dynamic aspects that are not within the scope of this dissertation. For the second goal, our tasks covered the major features of the tool, including searching, investigating, extracting a cluster as a call graph, and projecting to lower-level.

Table 22 shows the user study tasks with rationales. The first two tasks asked the participants to determine the entry and exit points of the current graph using the tool's

Table 22: Description of the Tasks for the User Study

ID	Task Description
T1	Name two packages that have a high fan-out with no fan-in
T2	Name two packages that have a high fan-in with no fan-out. **Rationale.** Analysis of dependencies between entities and assessing the quality of system design are essential to assist in software comprehension.
T3	Find any interesting path in the package level and project it to class level **Rationale.** Investigating the internal structure of an artefact is a comprehension task.
T4	Removing the class 'Transformation' in the 'model' package, - How many classes will be affected? - Name all affected class and which package they belong to **Rationale.** Impact analysis allows us to estimate how much of an impact such a code change would have on the system. It can also help estimate the effort that needs to be made to make such changes.
T5	The source code of SweetHome3D comes with two versions (applet and desktop), - Find all functions that are used in applet versions - Find all functions that are shared between these two versions **Rationale.** Investigating the functionality of (a part of) the system and understanding its role on the software is one of the main and useful activities in software comprehension for engineers and researchers.

filtering and search features. In the third task, the participants were asked to analyze and investigate the resulted clusters and project the cluster of interest using the projection feature. In the fourth task, the participants were asked to explore the information associated with a specific node. In the fifth task, the participants were asked to explore part of the system using a top-down approach supported by the tool.

6.5.4 Questionnaire

Our questionnaire consists of three types of questions including (1) Likert scale questions to evaluate the usefulness of the tool's features at different visualization views, (2) open-ended questions to gain feedback on the design of the tool, and (3) System Usability Score (SUS) based questions, to evaluate the usability of the tool. The list of questions and their types are listed in Table 23.

6.6 Results and Discussion

We first report the results of usefulness questions, Q1 to Q10, respectively, and then discuss the strengths and weaknesses of the tool from user feedback from Q11 and Q12. Finally, we report and discuss the results of the usability score.

6.6.1 Usefulness

To evaluate the usefulness of the tool, the participants were asked to perform a set of tasks and answer a survey with the following ten questions, and express their opinions accordingly. The results are illustrated using a Likert scale in Figure 40.

Q1: This question asks participants about their opinion on the tool's interface.

Table 23: User Study Questions and their Types

#	Question	Type
Q1	The tool's interface is intuitive and user-friendly	Likert Scale
Q2	I would recommend software developers to use this tool	Likert Scale
Q3	I would prefer to manually inspect the codebase rather than using this tool	Likert Scale
Q4	I believe that using this tool can save time and efforts in inspecting the codebase of a given system	Likert Scale
Q5	The task of identifying a feature that is implemented over multiple functions in the codebase is faster to achieve using this tool rather than manually inspecting the code	Likert Scale
Q6	Filtering by type, searching, retrieving the information, code coloring and shapes of the nodes were helpful during program comprehension tasks	Likert Scale
Q7	The generated clustered graphs are beneficial to identify the functionality structure of a given system–visually–without having to inspect the source code	Likert Scale
Q8	The multi-level call graphs visualization is beneficial to understand the overall of system structure from different views (functions, classes, and packages).	Likert Scale
Q9	The tool is useful for clustering and visually exploration the execution paths of a system in both views (i.e., call graph and hierarchical view)	Likert Scale
Q10	Overall, the visualization tool is useful to understand the software structure	Likert Scale
Q11	Please list any reasons for your answer to the previous question	Open-Ended
Q12	Would you like to add other comments? Limitations? Suggestions?	Open-Ended
Q1	I think that I would like to use this system frequently	Likert Scale
Q2	I found the system unnecessarily complex	Likert Scale
Q3	I thought the system was easy to use	Likert Scale
Q4	I think that I would need the support of a technical person to be able to use this system	Likert Scale
Q5	I found the various functions in this system were well integrated	Likert Scale
Q6	I thought there was too much inconsistency in this system	Likert Scale
Q7	I would imagine that most people would learn to use this system very quickly	Likert Scale
Q8	I found the system very cumbersome/awkward to use	Likert Scale
Q9	I felt very confident using the system	Likert Scale
Q10	I needed to learn a lot of things before I could get going with this system	Likert Scale

More than half of the participants, 12 out of 18, agree and strongly agree that the CodEx interface is intuitive and user-friendly.

Q2: We asked participants whether they would recommend CodEx to other developers. The responses were positive, and the vast majority agree with this statement, which represents 94.4% of the participants.

Q3: The goal of this question was to assess whether experienced developers (all of our participants have more than three years of experience using Java) would prefer manual inspection of the code over an automated tool that visualizes the code structure using call graphs. The results show that only one participant strongly agrees with this statement, 4 participants agree, while five disagree and one strongly disagrees.

While five participants illustrated that they would prefer to inspect the codebase manually rather than using our tool, upon further investigation and discussions with the participants, we discovered a discrepancy between our question and what participants understood from the question. The participants referred to the fact that they would prefer to inspect the source code directly for debugging and maintenance tasks, which does not contradict our tool. Our goal of the question was targeted towards the cognitive tasks of understanding the system's structure, but not the actual debugging task. Overall, low-level maintenance tools and high-level analysis tools (ours) are complementary in nature for supporting the understanding of software systems [137]. The answers from Q4 actually support this claim, as we explain in the following.

Q4: In this question, we aimed to assess the execution overhead of using CodEx. 16 out of 18 participants agree and strongly agree that using CodEx could save them time

and effort while carrying out the task of program comprehension. These findings strongly align with our execution time analysis that was discussed with the illustrative examples above.

Q5: This question focused on the specific task of identifying the functionality of a system under investigation. We asked the developers to compare carrying out this task using CodEx versus manual inspection. The results show that more than half, 61.1% prefer to inspect the system using CodEx. Thus, CodEx is useful to identify a feature in the system and all the underlying functions that implement it.

Q6: After asking the participants to explore different features of the system, we asked them whether or not these features were helpful for program comprehension tasks. We noticed that the majority, 14 out of 18, agree with this statement.

Q7: This specific question reflects the participants' opinion on the usability of generating and visualizing clusters without looking at the source code. Specifically, 13 out of 18 participants agree and strongly agree with this statement. Only two participants disagree with the statement.

Q8: In this question, we asked the participants if the multi-level call graphs are helpful to visually understand the system structure from the views of packages, classes, and functions. As expected, multi-level visualizing of the call graphs seems to be one of the best features that facilitate overall program comprehension. 10 participants agree, 6 participants strongly agree, and only 2 participants are neutral about this statement. None of the responses digressed with this statement.

Q9: In this question, we aimed to assess the overall satisfaction of the two main

contributions of our tool, i.e., multi-level and hierarchical visualizations of the call graphs. The responses to this statement included the following: 1 disagree, 2 neutral, 11 agree, and 4 strongly agree. Overall, about 80% of the participants agree and strongly agree that multi-level and hierarchical visualizations of the call graphs are useful to understand the overall structure of a software system.

Q10: In this question, the participants were asked to rate the overall usefulness of CodEx. The resulting scores show that 88.8% of the participants agreed, while 11% have neutral responses.

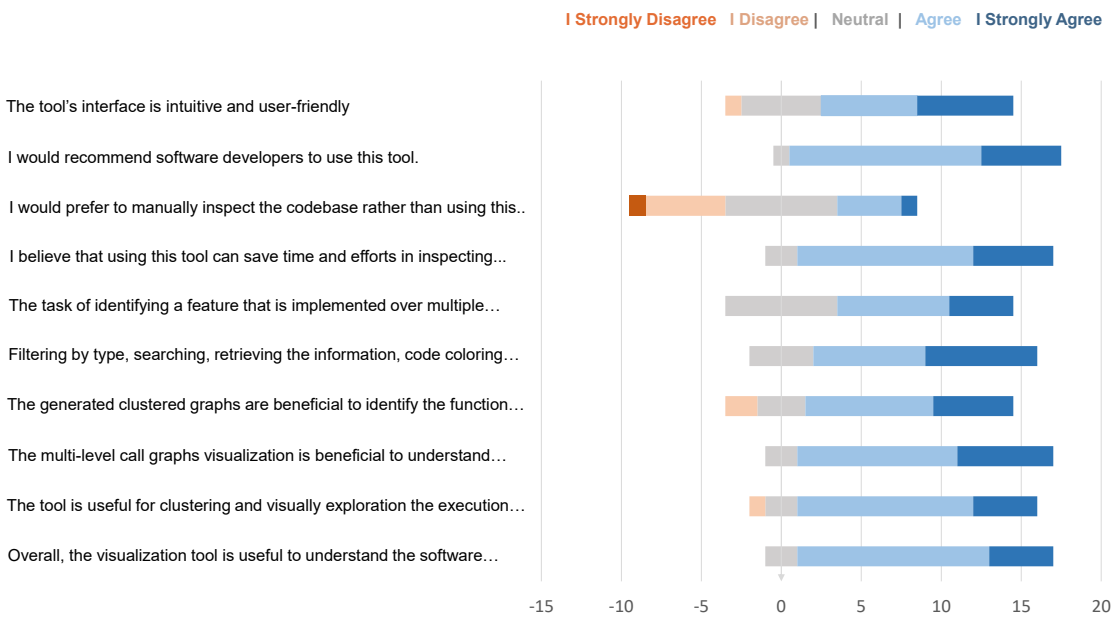


Figure 40: Likert Scale Representation of the Survey Responses.

6.6.2 Participants' Feedback

The questionnaire contains two open-end questions. The participants were asked to evaluate their overall opinion regarding the tool, provide us feedback that could help improve the tool and share valuable insights into the difficulties encountered during the analysis. We summarize and highlight participants' feedback as follows:

Graph Filtering. Some participants thought that it was challenging to explore the function call graph. One participant commented: "The visualization seems slow and hangs when there are too many nodes and edges." It is one of the major drawbacks of node-link representation. It tends to become highly cluttered when large numbers of nodes and edges are visualized. Several participants suggested an improvement by providing features to filter/hide irrelevant nodes and edges, which allow the graph to be more manageable and easy to explore. Some participants suggest that the tool may be more selectively showing only execution paths containing a particular node.

Lack of Information and Customization. We used some basic software metrics such as in-degree, out-degree to help identify components that, for instance, need to be refactored. However, two participants suggested including more structural data. One participant commented: "It would be good to know how many classes there are in a package." Some participants suggested "providing more software metrics data, such as LOC, WMC, and there should be ways to query and sort, such as by color or size." Another suggestion was made regarding the ability to customize colors or styles of graphs. One participant suggested adding a brief description that appeared when hovering over an icon. Another participant suggested an enrichment tool available as a plugin for the

popular IDEs, including IntelliJ and VS.Code.

6.6.3 Usability

To analyze the usability of the tool, we considered The System Usability Score (SUS). Figure 41 shows the average value for the System Usability Scores, which is 72.6. According to a previous study, [76], this score is above the average 68, which can be interpreted as Good. Although the tool has a 'Good' average, the minimum score is 52 among other similar studies.

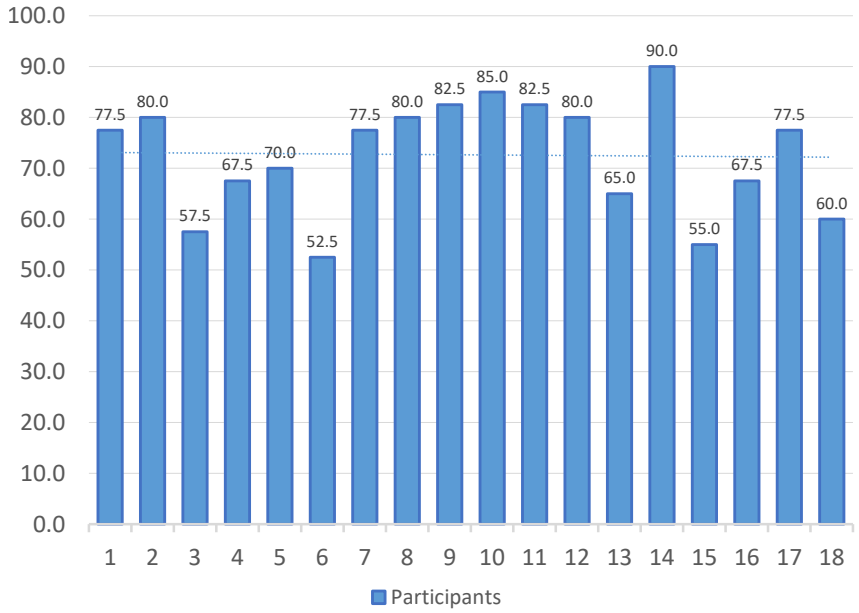


Figure 41: The System Usability Score

6.6.4 Threats to Validity

One possible threat to validity is that our tasks may not be designed to be representative software comprehension tasks, and they are biased favoring the tool. However,

in our defense, we had used an established comprehension task framework introduced by Pacione et al [99]. They studied several sets of tasks used in comprehension evaluation literature and software visualization. We designed our tasks to cover most comprehension activities based on Pacione's framework. The uncovered activities are mainly concerned with dynamic aspects that we do not consider in our study.

Another possible threat to validity is that the number of subjects in the study is less than the usual–only one subject system. However, we argue that we used a system that is well-curated and well-documented of manageable size, which we believe is a good representative of real software systems. Moreover, a primary concern was to validate the usefulness of the tool. Therefore, we needed to be able to check the subject system manually. With our tool implemented and ready to be used, one can conduct several other use cases, and it is made our future work.

CHAPTER 7

CONCLUSION AND FEATURE WORK

The main intent of this dissertation is to support program comprehension using visual analysis and exploration of a software system. Moreover, to assist software developers in understanding the software system from a high level of abstraction to a low level of implementation with the ability to focus on particular parts of the system individually. We highlight some of the significant research contributions of the work described in this dissertation:

- Develop an automated language-independent approach for analyzing the source code of a system and visualizing its structure in multi-level abstractions using the call graph. This allows the developers to understand the implementation and the structure of a software system using the call graph at multi-level of abstractions.

- Develop an automated data-driven approach that can automatically map high-level system functionality to its low-level using our proposed clustering techniques. This helps the developer bridges the cognitive gap between the system's overall functionality and its implementation.

- Develop an interactive clustering-based visualization tool to facilitate the process of analysis, exploration, and mapping of resulted clusters to the overall system structure. This allows the developers to visually analyze and interpret the clustered execution paths by inking the clustering results to the overall system structure.

- To interpret the meaning of the resulted clusters, we utilize topic modeling techniques to understand clusters' functionality by generating labels that reflect the meaning of each cluster.

The user study that we conducted shows that some of the participants had comments regarding the tool's design and appearance. Although the majority of them expressed their interest, few of them suggest more improvement such as extracting and calculating more software metrics and utilizing the size, shape, and color of a node or edge to reflect these matrices to give the developer more insight into the complexity of the software. Thus, our future work will be focused on the issues raised by the participants and plan to resolve them by adding those new functionalities. We also plan to use GPU platforms to further reduce the execution time spent on the analysis tasks such as clustering technique.

APPENDIX A

GENERATED CALL GRAPHS

In this chapter, we present all the generated function call graphs of six case studies that were conducted in Chapter 2 using our tool.

Table 24: Structure Analysis of the Function Call Graph for Each Case Study

Entity	Detectron	Flask	Keras	PHNotepad	SweetHome 3D	WEKA
Nodes	525	370	1,779	36	5,148	14,742
Edges	740	360	2,347	36	9,501	35,575
Entry Point	108	123	844	4	1,517	5,031
Exit Point	207	168	591	28	2,821	4,982
Articulation Point	133	115	376	8	577	2,460

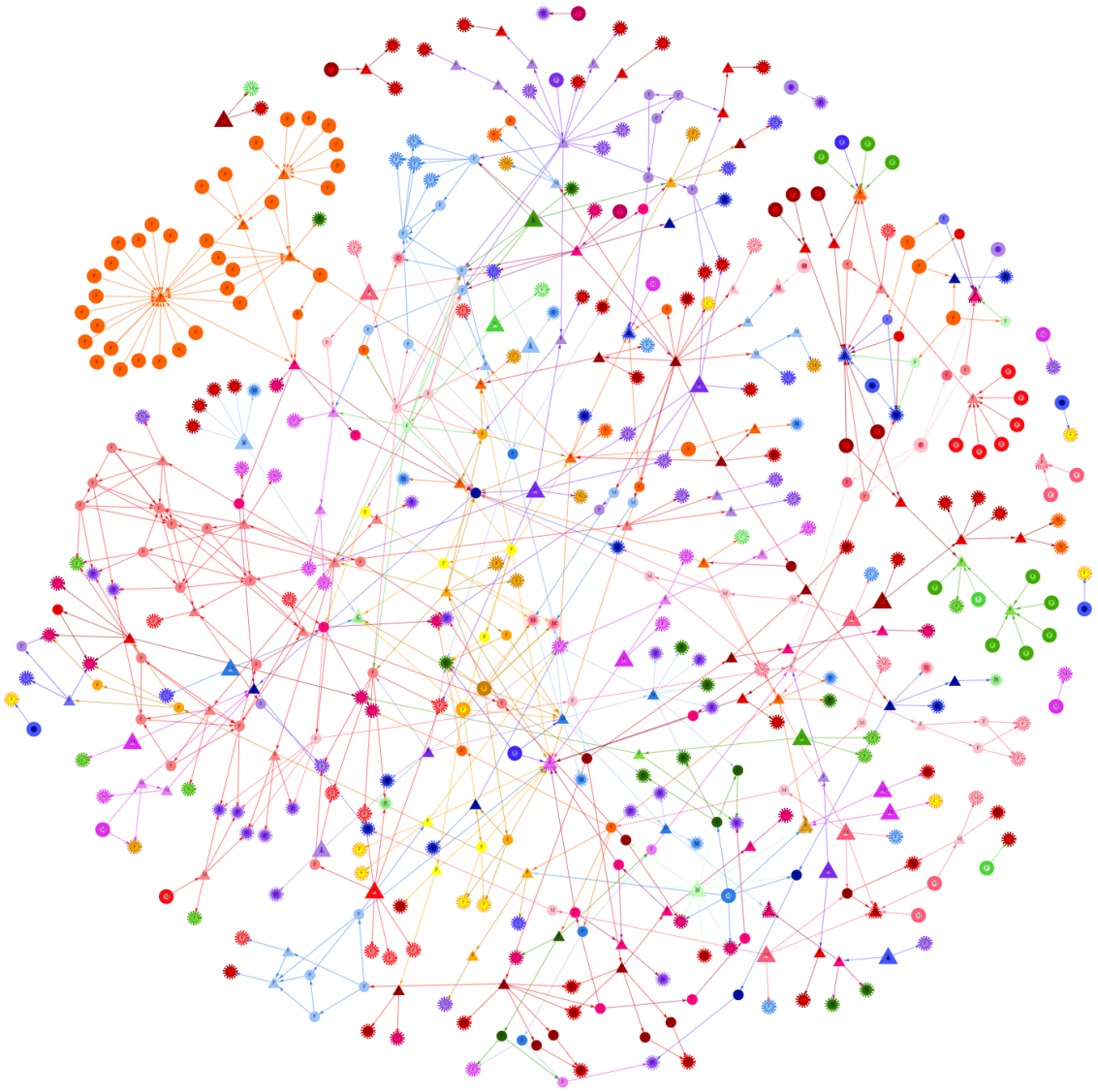


Figure 42: Detectron Function Call Graph

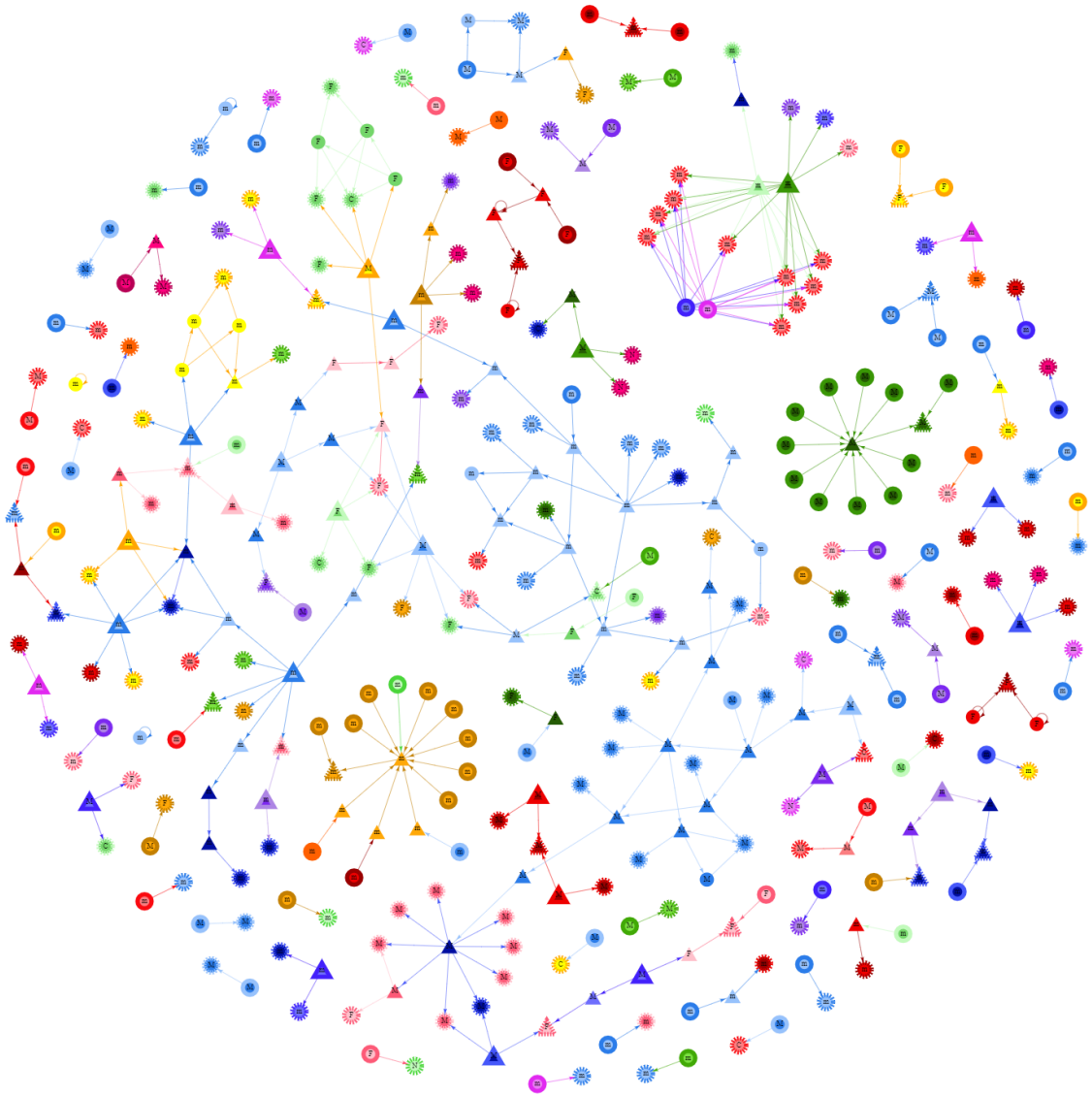


Figure 43: Flask Function Call Graph

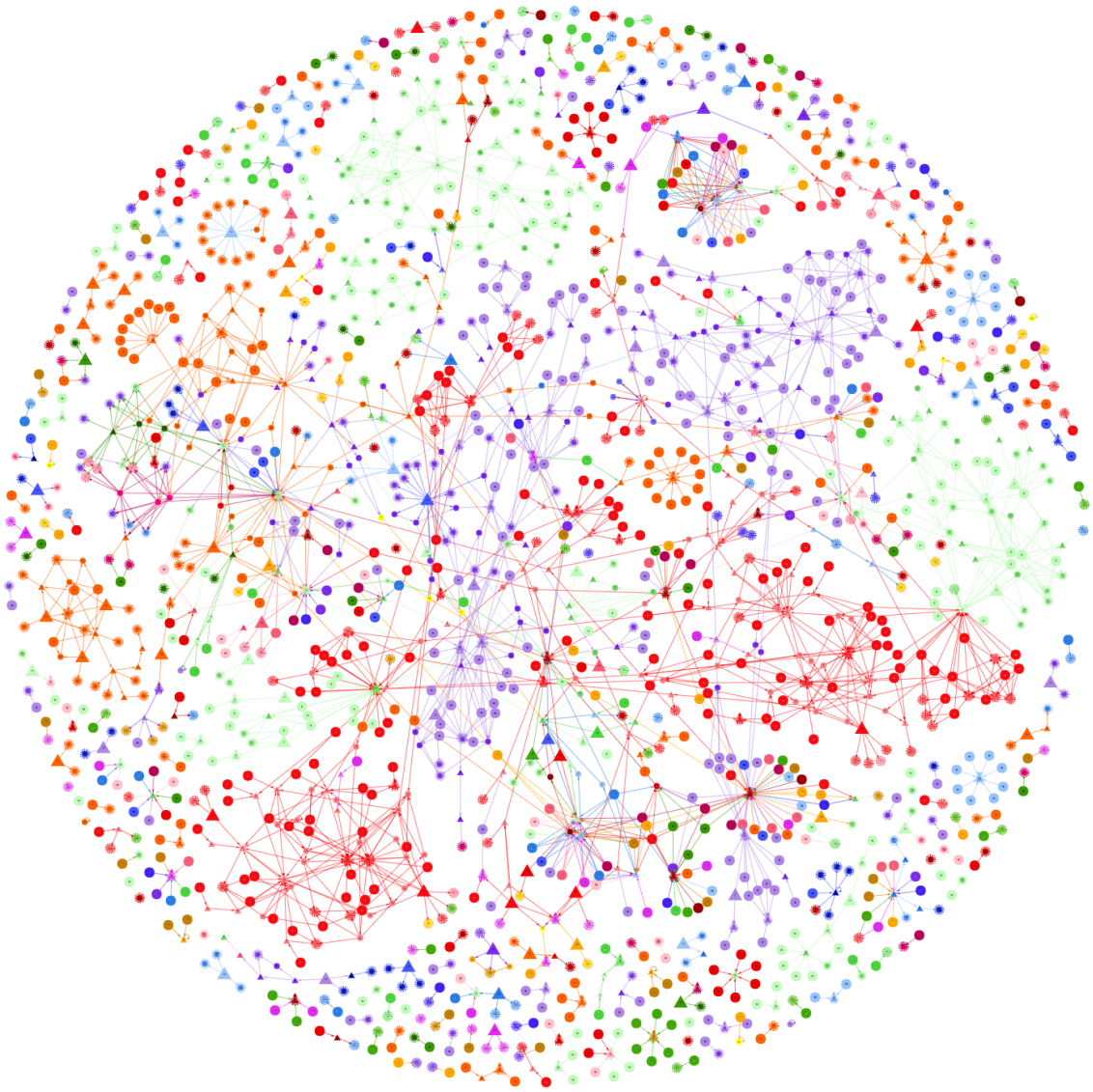


Figure 44: Keras Function Call Graph

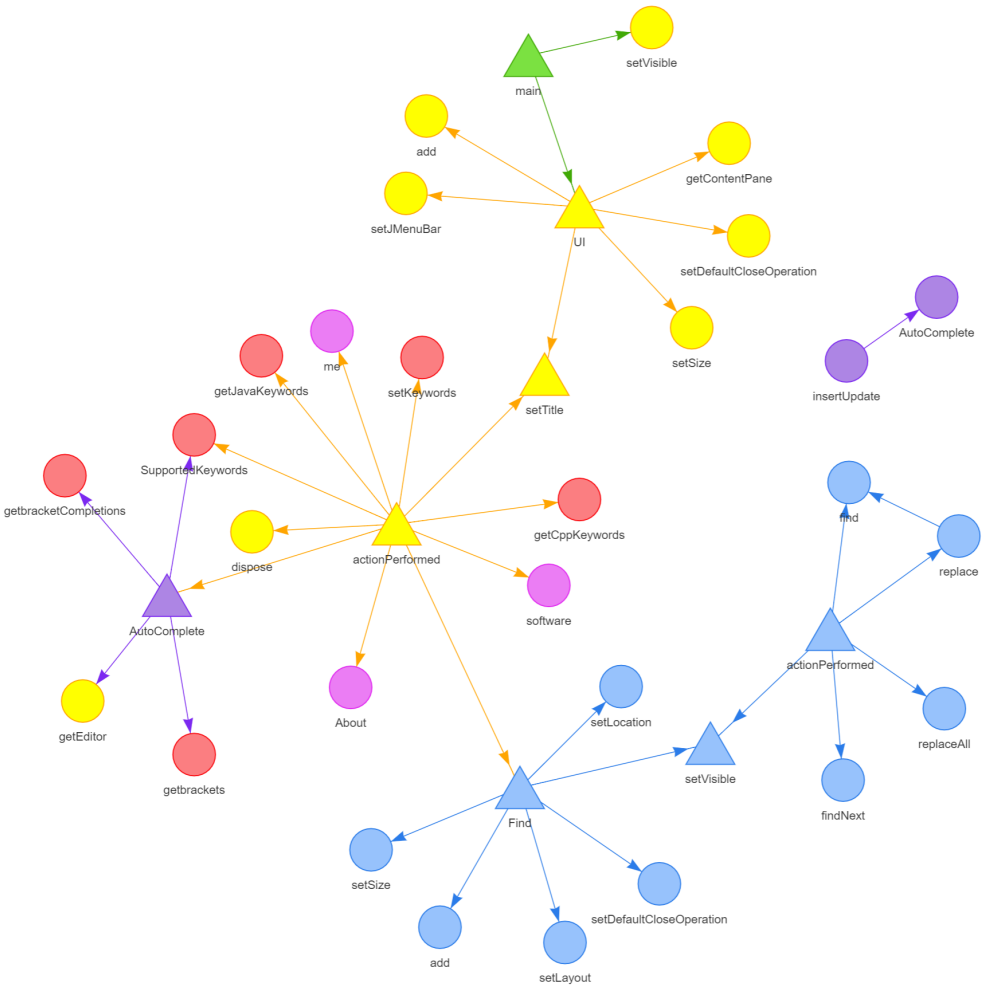


Figure 45: PHNotepad Function Call Graph

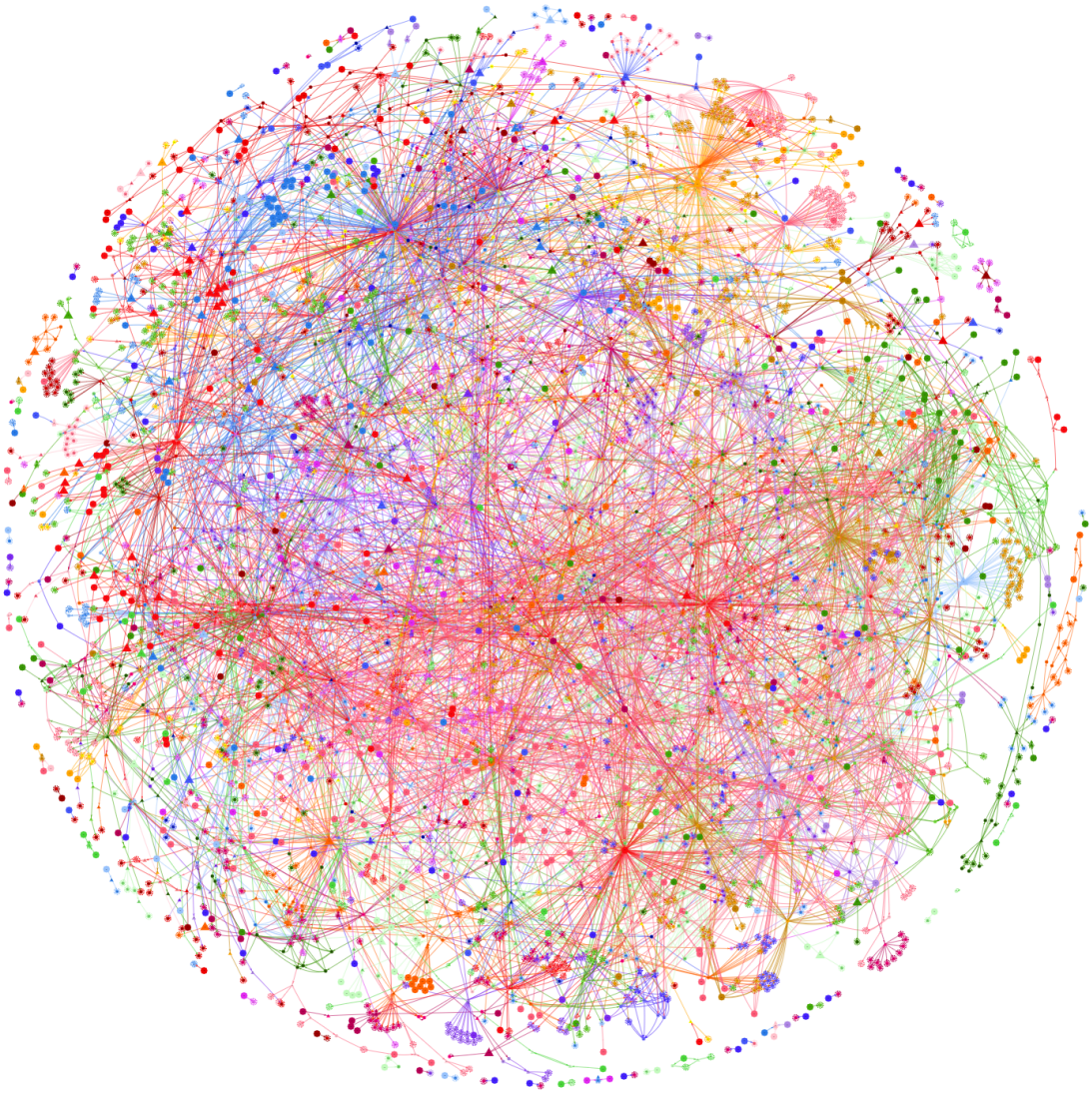


Figure 46: SweetHome3D Function Call Graph

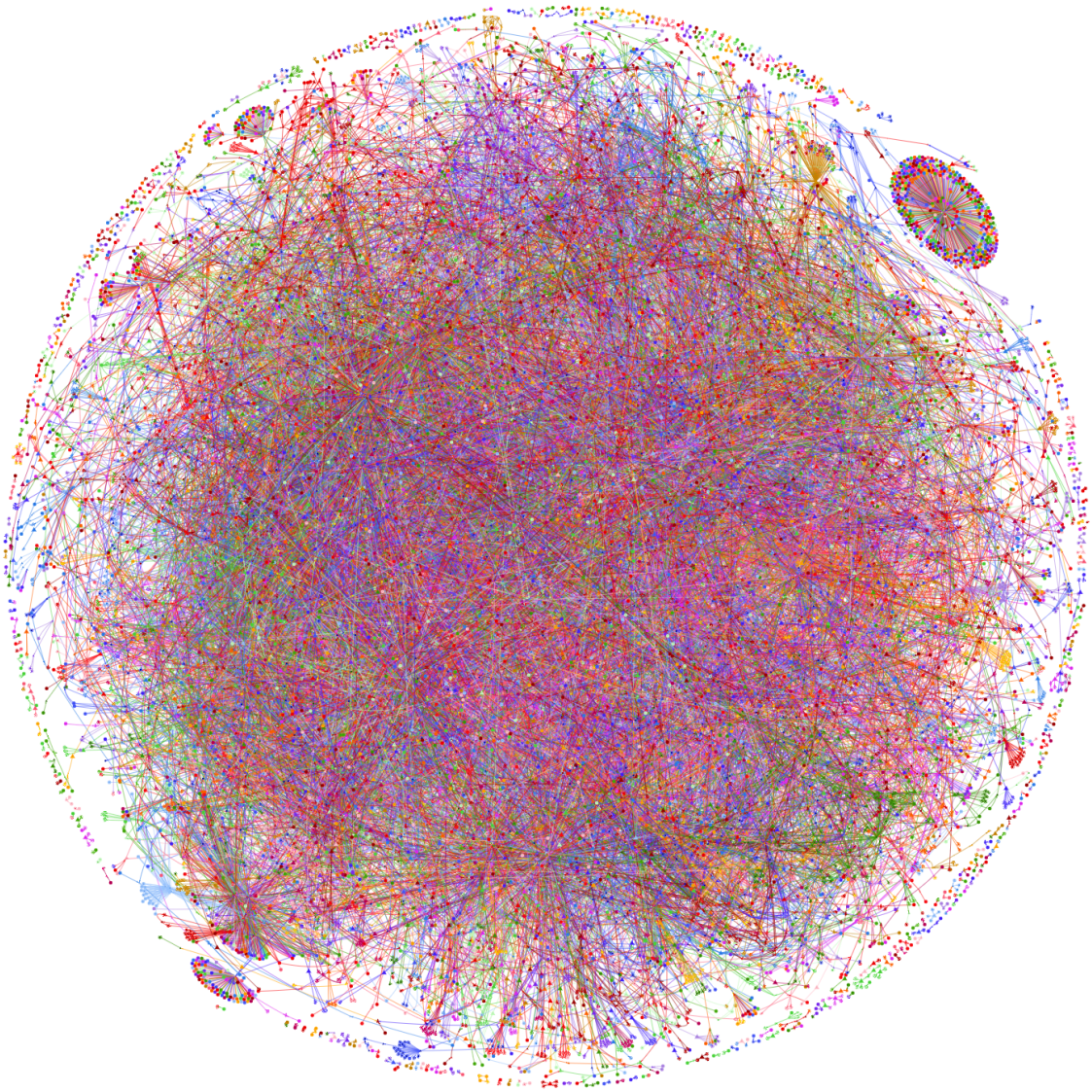


Figure 47: WEKA Function Call Graph

REFERENCE LIST

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 265–283.

[2] Alanazi, R., Gharibi, G., and Lee, Y. Facilitating program comprehension with call graph multilevel hierarchical abstractions. *Journal of Systems and Software 176* (2021), 110945.

[3] ALI, V. T., and Aksoy TUYSUZ, M. A. Analysis of function-call graphs of open-source software systems using complex network analysis. *Pamukkale University Journal of Engineering Sciences 26*, 2 (2020).

[4] Alnabhan, M., Hammouri, A., Hammad, M., Atoum, M., and Al-Thnebat, O. 2D visualization for object-oriented software systems. In *2018 International Conference on Intelligent Systems and Computer Vision (ISCV)* (2018), IEEE, pp. 1–6.

[5] Andritsos, P., and Tzerpos, V. Information-theoretic software clustering. *IEEE Transactions on Software Engineering 31*, 2 (2005), 150–165.

[6] Anquetil, N., and Lethbridge, T. C. Experiments with clustering as a software remodularization method. In *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)* (1999), IEEE, pp. 235–255.

[7] Askari, S. A novel and fast MIMO fuzzy inference system based on a class of fuzzy clustering algorithms with interpretability and complexity analysis. *Expert Systems with Applications 84* (2017), 301–322.

[8] Aung, T. W. W., Huo, H., and Sui, Y. A Literature Review of Automatic Traceability Links Recovery for Software Change Impact Analysis. In *Proceedings of the 28th International Conference on Program Comprehension* (2020), pp. 14–24.

[9] Ball, T. The concept of dynamic analysis. In *Software Engineering—ESEC/FSE'99* (1999), Springer, pp. 216–234.

[10] Bauer, M., and Trifu, M. Architecture-aware adaptive clustering of OO systems. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* (2004), IEEE, pp. 3–14.

[11] Berkhin, P. A survey of clustering data mining techniques. In *Grouping multidimensional data*. Springer, 2006, pp. 25–71.

[12] Bhattacharya, P., Iliofotou, M., Neamtiu, I., and Faloutsos, M. Graph-based analysis and prediction for software evolution. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 419–429.

[13] Bird, S. Proceedings of the COLING/ACL on Interactive presentation sessions, 2006.

[14] Bogar, A. M., Lyons, D. M., and Baird, D. Lightweight Call-Graph Construction for Multilingual Software Analysis. *arXiv preprint arXiv:1808.01213* (2018).

[15] Bohnet, J., and Döllner, J. Visual exploration of function call graphs for feature location in complex software systems. In *Proceedings of the 2006 ACM symposium on Software visualization* (2006), pp. 95–104.

[16] Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics 5* (2017), 135–146.

[17] Brito, R., Brito, A., Brito, G., and Valente, M. T. GoCity: code city for go. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2019), IEEE, pp. 649–653.

[18] Brooke, J. SUS: a "quick and dirty'usability. *Usability evaluation in industry* (1996), 189.

[19] Burkhardt, J.-M., Détienne, F., and Wiedenbeck, S. Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering 7*, 2 (2002), 115–156.

[20] Chan, A., Holmes, R., Murphy, G. C., and Ying, A. T. Scaling an object-oriented system execution visualizer through sampling. In *11th IEEE International Workshop on Program Comprehension, 2003.* (2003), IEEE, pp. 237–244.

[21] Chen, T.-H., Thomas, S. W., and Hassan, A. E. A survey on the use of topic models when mining software repositories. *Empirical Software Engineering 21*, 5 (2016), 1843–1919.

[22] Chen, T.-H., Thomas, S. W., Hemmati, H., Nagappan, M., and Hassan, A. E. An empirical study on the effect of testing on code quality using topic models: A case study on software development systems. *IEEE Transactions on Reliability 66*, 3 (2017), 806–824.

[23] Chevalier, C., and Pellegrini, F. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel computing 34*, 6-8 (2008), 318–331.

[24] Chevalier, C., and Safro, I. Comparison of coarsening schemes for multilevel graph partitioning. In *International Conference on Learning and Intelligent Optimization* (2009), Springer, pp. 191–205.

[25] Chollet, F., et al. Keras, 2015.

[26] Chong, C. Y., Lee, S. P., and Ling, T. C. Efficient software clustering technique using an adaptive and preventive dendrogram cutting approach. *Information and Software Technology 55*, 11 (2013), 1994–2012.

[27] Corbi, T. A. Program understanding: Challenge for the 1990s. *IBM Systems Journal 28*, 2 (1989), 294–306.

[28] Cornelissen, B., Holten, D., Zaidman, A., Moonen, L., Van Wijk, J. J., and Van Deursen, A. Understanding execution traces using massive sequence and circular bundle views. In *15th IEEE International Conference on Program Comprehension (ICPC'07)* (2007), IEEE, pp. 49–58.

[29] Cornelissen, B., Zaidman, A., Van Deursen, A., Moonen, L., and Koschke, R. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering 35*, 5 (2009), 684–702.

[30] Davey, J., and Burd, E. Evaluating the suitability of data clustering for software remodularisation. In *Proceedings Seventh Working Conference on Reverse Engineering* (2000), IEEE, pp. 268–276.

[31] Davis, T. A., Hager, W. W., Kolodziej, S. P., and Yeralan, S. N. Algorithm 1003: Mongoose, a Graph Coarsening and Partitioning Library. *ACM Transactions on Mathematical Software (TOMS) 46*, 1 (2020), 1–18.

[32] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. Indexing by latent semantic analysis. *Journal of the American society for information science 41*, 6 (1990), 391–407.

[33] Dit, B., Guerrouj, L., Poshyvanyk, D., and Antoniol, G. Can better identifier splitting techniques help feature location? In *2011 IEEE 19th International Conference on Program Comprehension* (2011), IEEE, pp. 11–20.

[34] Ellson, J., Gansner, E., Koutsofios, L., North, S. C., and Woodhull, G. Graphviz—open source graph drawing tools. In *International Symposium on Graph Drawing* (2001), Springer, pp. 483–484.

[35] eTeks. SweetHome 3D. `http://www.sweethome3d.com`.

[36] Exton, C. Constructivism and program comprehension strategies. In *Proceedings 10th International Workshop on Program Comprehension* (2002), IEEE, pp. 281–284.

[37] Fairley, R. E. Tutorial: Static analysis and dynamic testing of computer software. *Computer 11*, 4 (1978), 14–23.

[38] Feng, Y., Dreef, K., Jones, J. A., and van Deursen, A. Hierarchical abstraction of execution traces for program comprehension. In *Proceedings of the 26th Conference on Program Comprehension* (2018), ACM, pp. 86–96.

[39] Frank, E., and Mark, A. Hall, and Ian H. Witten (2016). The WEKA Workbench. Online Appendix for" Data Mining: Practical Machine Learning Tools and Techniques, 2016.

[40] Gansner, E., and Ellson, J. Graphviz-Graph Visualization Software. *URL http://www.graphviz. org/.(Cited on pages 69, 81, and 82.)* (2017).

[41] Gharibi, G., Alanazi, R., and Lee, Y. Automatic Hierarchical Clustering of Static Call Graphs for Program Comprehension. In *2018 IEEE International Conference on Big Data (Big Data)* (2018), IEEE, pp. 4016–4025.

[42] Gharibi, G., Tripathi, R., and Lee, Y. Code2graph: automatic generation of static call graphs for Python source code. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (2018), ACM, pp. 880–883.

[43] Girshick, R., Radosavovic, I., Gkioxari, G., Dollár, P., and He, K. Detectron. https://github.com/facebookresearch/detectron, 2018.

[44] Gousios, G. java-callgraph: Java Call Graph Utilities. https://github.com/gousiosg/java-callgraph, 2019.

[45] Graham, S. L., Kessler, P. B., and Mckusick, M. K. Gprof: A call graph execution profiler. In *ACM Sigplan Notices* (1982), vol. 17, ACM, pp. 120–126.

[46] Grinberg, M. *Flask web development: developing web applications with python*. " O'Reilly Media, Inc.", 2018.

[47] Grove, D., DeFouw, G., Dean, J., and Chambers, C. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices 32*, 10 (1997), 108–124.

[48] Hagberg, A., Swart, P., and S Chult, D. Exploring network structure, dynamics, and function using NetworkX. Tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[49] Hahsler, M., and Piekenbrock, M. *dbscan: Density Based Clustering of Applications with Noise (DBSCAN) and Related Algorithms*, 2021. R package version 1.1-8.

[50] Hahsler, M., Piekenbrock, M., and Doran, D. dbscan: Fast Density-Based Clustering with R. *Journal of Statistical Software 91*, 1 (2019), 1–30.

[51] Hamou-Lhadj, A., Braun, E., Amyot, D., and Lethbridge, T. Recovering behavioral design models from execution traces. In *Ninth European Conference on Software Maintenance and Reengineering* (2005), IEEE, pp. 112–121.

[52] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del R'ıo, J. F., Wiebe, M., Peterson, P., G'erard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with NumPy. *Nature 585*, 7825 (Sept. 2020), 357–362.

[53] Hartigan, J. A., and Wong, M. A. AK-means clustering algorithm. *Journal of the Royal Statistical Society: Series C (Applied Statistics) 28*, 1 (1979), 100–108.

[54] Hassan, A. E., Hindle, A., Runeson, P., Shepperd, M., Devanbu, P., and Kim, S. Roundtable: What's next in software analytics. *IEEE software 30*, 4 (2013), 53–56.

[55] Hendrickson, B., and Leland, R. W. A Multi-Level Algorithm For Partitioning Graphs. *SC 95*, 28 (1995), 1–14.

[56] Henry, P. Notepad. `https://github.com/pH-7/Simple-Java-Text-Editor`, 2021.

[57] Hoogendorp, H. *Extraction and visual exploration of call graphs for large software systems*. PhD thesis, Faculty of Science and Engineering, 2010.

[58] Hopcroft, J., and Tarjan, R. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM 16*, 6 (1973), 372–378.

[59] Horner, E. Constructing Call Graphs.

[60] Islam, S., Krinke, J., Binkley, D., and Harman, M. Coherent clusters in source code. *Journal of Systems and Software 88* (2014), 1–24.

[61] Jain, A. K. Data clustering: 50 years beyond K-means. *Pattern recognition letters 31*, 8 (2010), 651–666.

[62] Jin, W., Cai, Y., Kazman, R., Zheng, Q., Cui, D., and Liu, T. ENRE: a tool framework for extensible eNtity relation extraction. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings* (2019), IEEE Press, pp. 67–70.

[63] Jin, W., Liu, T., Zheng, Q., Cui, D., and Cai, Y. Functionality-Oriented Microservice Extraction Based on Execution Trace Clustering. In *2018 IEEE International Conference on Web Services (ICWS)* (2018), IEEE, pp. 211–218.

[64] Kargar, M., Isazadeh, A., and Izadkhah, H. Semantic-based software clustering using hill climbing. In *2017 International Symposium on Computer Science and Software Engineering Conference (CSSE)* (2017), IEEE, pp. 55–60.

[65] Karypis, G., and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing 20*, 1 (1998), 359–392.

[66] Khaloo, P., Maghoumi, M., Taranta, E., Bettner, D., and Laviola, J. Code park: A new 3d code visualization tool. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)* (2017), IEEE, pp. 43–53.

[67] Khan, M. *Supporting Source Code Feature Analysis Using Execution Trace Mining*. PhD thesis, University of Saskatchewan, 2013.

[68] Khan, M. S. A Topic Modeling approach for Code Clone Detection. Master's thesis, University of North Florida. School of Computing, 2019.

[69] Kosara, R., Healey, C., Interrante, V., Laidlaw, D., and Ware, C. User studies: Why, how, and when? *IEEE Computer Graphics and Applications 23*, 4 (July 2003), 20–25.

[70] Kuhn, A., Ducasse, S., and Gírba, T. Semantic clustering: Identifying topics in source code. *Information and Software Technology 49*, 3 (2007), 230–243.

[71] Kuhn, A., Greevy, O., and Gırba, T. Applying semantic analysis to feature execution traces. In *Proceedings of Workshop on Program Comprehension through Dynamic Analysis (PCODA 2005)* (2005), pp. 48–53.

[72] Kulkarni, A., and Pedersen, T. SenseClusters: unsupervised clustering and labeling of similar contexts. In *Proceedings of the ACL Interactive Poster and Demonstration Sessions* (2005), pp. 105–108.

[73] LaToza, T. D., and Myers, B. A. Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2011), IEEE, pp. 117–124.

[74] Lemos, O. A. L., Zanichelli, F. C., Rigatto, R., Ferrari, F., and Ghosh, S. Visualization, Analysis, and Testing of Java and AspectJ Programs with Multi-level System Graphs. In *2013 27th Brazilian Symposium on Software Engineering* (2013), Ieee, pp. 49–58.

[75] Letovsky, S. Cognitive processes in program comprehension. *Journal of Systems and software 7*, 4 (1987), 325–339.

[76] Lewis, J. R., and Sauro, J. Item benchmarks for the system usability scale. *Journal of Usability Studies 13*, 3 (2018).

[77] Li, K. *Combining Static and Dynamic Analysis for Bug Detection and Program Understanding*. PhD thesis, University of Massachusetts Amherst, 2016.

[78] Li, P., Hastie, T. J., and Church, K. W. Very sparse random projections. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* (2006), pp. 287–296.

[79] Li, Q., Guindani, M., Reich, B. J., Bondell, H. D., and Vannucci, M. A Bayesian mixture model for clustering and selection of feature occurrence rates under mean constraints. *Statistical Analysis and Data Mining: The ASA Data Science Journal 10*, 6 (2017), 393–409.

[80] Maalej, W., Tiarks, R., Roehm, T., and Koschke, R. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM) 23*, 4 (2014), 31.

[81] Mahmoud, A., and Bradshaw, G. Semantic topic models for source code analysis. *Empirical Software Engineering 22*, 4 (2017), 1965–2000.

[82] Maqbool, O., and Babri, H. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering 33*, 11 (2007), 759–780.

[83] Maqbool, O., and Babri, H. A. The weighted combined algorithm: A linkage algorithm for software clustering. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* (2004), IEEE, pp. 15–24.

[84] Maskeri, G., Sarkar, S., and Heafield, K. Mining business topics in source code using latent dirichlet allocation. In *Proceedings of the 1st India software engineering conference* (2008), pp. 113–120.

[85] Medvidovic, N., and Jakobac, V. Using software evolution to focus architectural recovery. *Automated Software Engineering 13*, 2 (2006), 225–256.

[86] Mens, T., Serebrenik, A., and Cleve, A. *Evolving Software Systems*, vol. 190. Springer, 2014.

[87] Merino, L., Ghafari, M., and Nierstrasz, O. Towards actionable visualization for software developers. *Journal of software: evolution and process 30*, 2 (2018), e1923.

[88] Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[89] Misiak, M., Schreiber, A., Fuhrmann, A., Zur, S., Seider, D., and Nafeie, L. Islandviz: a tool for visualizing modular software systems in virtual reality. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)* (2018), IEEE, pp. 112–116.

[90] Mitchell, B. S., and Mancoridis, S. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering 32*, 3 (2006), 193–208.

[91] Mkaouer, M. W., Kessentini, M., Bechikh, S., Cinnéide, M. Ó., and Deb, K. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering 21*, 6 (2016), 2503–2545.

[92] Mueller, A. WordCloud. `https://amueller.github.io/word_cloud/index.html`, 2021.

[93] Muhammad, S., Maqbool, O., and Abbasi, A. Q. Evaluating relationship categories for clustering object-oriented software systems. *IET software 6*, 3 (2012), 260–274.

[94] Murphy, G. C., Notkin, D., Griswold, W. G., and Lan, E. S. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology (TOSEM) 7*, 2 (1998), 158–191.

[95] Naeimian, A. Parallel Paths Analysis Using Function Call Graphs. Master's thesis, University of Waterloo, 2019.

[96] Naseem, R., Maqbool, O., and Muhammad, S. Cooperative clustering for software modularization. *Journal of Systems and Software 86*, 8 (2013), 2045–2062.

[97] Noack, A. Energy models for graph clustering. *J. Graph Algorithms Appl. 11*, 2 (2007), 453–480.

[98] Oliphant, T. E. *A guide to NumPy*, vol. 1. Trelgol Publishing USA, 2006.

[99] Pacione, M. J., Roper, M., and Wood, M. A novel software visualisation model to support software comprehension. In *11th working conference on reverse engineering* (2004), IEEE, pp. 70–79.

[100] Parr, T. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[101] Paul, A. K., and Shill, P. C. New automatic fuzzy relational clustering algorithms using multi-objective NSGA-II. *Information Sciences 448* (2018), 112–133.

[102] Rajlich, V., and Wilde, N. The role of concepts in program comprehension. In *Proceedings 10th International Workshop on Program Comprehension* (2002), IEEE, pp. 271–278.

[103] Reddivari, S., and Bhowmik, T. A Qualitative Investigation of Landmarks in Software Code Navigation. In *Proceedings of the 2nd International Conference on Compute and Data Analysis* (2018), pp. 80–84.

[104] Reddivari, S., and Kotapalli, M. On the use of visual clustering to identify landmarks in code navigation. In *2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)* (2017), IEEE, pp. 219–228.

[105] Reddivari, S., Kotapalli, M., and Niu, N. SDVisu: A tool for clustering-based visual exploration of static dependencies. In *2017 Computing Conference* (2017), IEEE, pp. 1373–1374.

[106] Reese, R., and Johnson, J. *jMonkeyEngine 3.0 game development: A practical guide*. P8tech, 2015.

[107] Řehůřek, R., and Sojka, P. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks* (Valletta, Malta, May 2010), ELRA, pp. 45–50. `http://is.muni.cz/publication/884893/en`.

[108] Rehurek, R., and Sojka, P. Gensim–python framework for vector space modelling. *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic 3*, 2 (2011).

[109] Reimers, N., and Gurevych, I. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing* (11 2019), Association for Computational Linguistics.

[110] Reiss, S. P. Dynamic detection and visualization of software phases. *ACM SIGSOFT Software Engineering Notes 30*, 4 (2005), 1–6.

[111] Reiss, S. P., and Renieris, M. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001* (2001), IEEE, pp. 221–230.

[112] Ryder, B. G. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, 3 (1979), 216–226.

[113] Saeed, M., Maqbool, O., Babri, H. A., Hassan, S. Z., and Sarwar, S. M. Software clustering techniques and the use of combined algorithm. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.* (2003), IEEE, pp. 301–306.

[114] Salis, V., Sotiropoulos, T., Louridas, P., Spinellis, D., and Mitropoulos, D. PyCG: Practical Call Graph Generation in Python. *nature 15* (2020), 16.

[115] Sander, J., Qin, X., Lu, Z., Niu, N., and Kovarsky, A. Automatic extraction of clusters from hierarchical clustering representations. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining* (2003), Springer, pp. 75–87.

[116] Santos, G., Valente, M. T., and Anquetil, N. Remodularization analysis using semantic clustering. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)* (2014), IEEE, pp. 224–233.

[117] Savage, T., Dit, B., Gethers, M., and Poshyvanyk, D. Topic XP: Exploring topics in source code using latent Dirichlet allocation. In *2010 IEEE International Conference on Software Maintenance* (2010), IEEE, pp. 1–6.

[118] Schröter, I., Krüger, J., Siegmund, J., and Leich, T. Comprehending studies on program comprehension. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)* (2017), IEEE, pp. 308–311.

[119] Shah, M. D., and Guyer, S. Z. An interactive microarray call-graph visualization. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)* (2016), IEEE, pp. 86–90.

[120] Shneiderman, B., and Mayer, R. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences 8*, 3 (1979), 219–238.

[121] Slob, G.-J., Dalpiaz, F., Brinkkemper, S., and Lucassen, G. The interactive narrator tool: Effective requirements exploration and discussion through visualization. In *Joint Proceedings of REFSQ-2018 Workshops, Doctoral Symposium, Live Studies*

Track, and Poster Track co-located with the 23rd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2018), Utrecht, The Netherlands, March 19, 2018. (2018).

[122] Spinellis, D. *Code reading: the open source perspective*. Addison-Wesley Professional, 2003.

[123] Srinivasa-Desikan, B. *Natural Language Processing and Computational Linguistics: A practical guide to text analysis with Python, Gensim, spaCy, and Keras*. Packt Publishing Ltd, 2018.

[124] Storey, M.-A. Theories, methods and tools in program comprehension: Past, present and future. In *13th International Workshop on Program Comprehension (IWPC'05)* (2005), IEEE, pp. 181–191.

[125] Sun, X., Liu, X., Li, B., Li, B., Lo, D., and Liao, L. Clustering classes in packages for program comprehension. *Scientific Programming 2017* (2017).

[126] Thomas, S. W., Adams, B., Hassan, A. E., and Blostein, D. Modeling the evolution of topics in source code histories. In *Proceedings of the 8th working conference on mining software repositories* (2011), pp. 173–182.

[127] Tian, K., Revelle, M., and Poshyvanyk, D. Using latent dirichlet allocation for automatic categorization of software. In *2009 6th IEEE International Working Conference on Mining Software Repositories* (2009), IEEE, pp. 163–166.

[128] Turhan, B., Kocak, G., and Bener, A. Data mining source code for locating software bugs: A case study in telecommunication industry. *Expert Systems with Applications 36*, 6 (2009), 9986–9990.

[129] Van der Maaten, L., and Hinton, G. Visualizing data using t-SNE. *Journal of machine learning research 9*, 11 (2008).

[130] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Jarrod Millman, K., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C., Polat, l., Feng, Y., Moore, E. W., Vand erPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and Contributors, S. . . SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* (2020).

[131] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods 17* (2020), 261–272.

[132] von Mayrhauser, A., and Vans, A. M. Comprehension processes during large scale maintenance. In *Proceedings of 16th International Conference on Software Engineering* (1994), IEEE, pp. 39–48.

[133] Walshaw, C., and Cross, M. JOSTLE: parallel multilevel graph-partitioning software–an overview. *Mesh partitioning techniques and domain decomposition techniques* (2007), 27–58.

[134] Walunj, V., Gharibi, G., Ho, D. H., and Lee, Y. GraphEvo: Characterizing and Understanding Software Evolution using Call Graphs. In *2019 IEEE International Conference on Big Data (Big Data)* (2019), IEEE, pp. 4799–4807.

[135] Wang, T., and Liu, Y. Facilitating Scenario-Based Program Comprehension with Topic Models. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)* (2017), IEEE, pp. 642–647.

[136] Wettel, R. Visual exploration of large-scale evolving software. In *2009 31st International Conference on Software Engineering-Companion Volume* (2009), IEEE, pp. 391–394.

[137] Wettel, R. *Software systems as cities*. PhD thesis, Università della Svizzera italiana, 2010.

[138] Wiese, A., Ho, V., and Hill, E. A comparison of stemmers on source code identifiers for software search. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)* (2011), IEEE, pp. 496–499.

[139] Wiggerts, T. A. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering* (1997), IEEE, pp. 33–43.

[140] Wilde, N. Faster reuse and maintenance using software reconnaissance. *Technical Report SERC-TR-75F* (1994).

[141] Wu, H. C., Luk, R. W. P., Wong, K. F., and Kwok, K. L. Interpreting tf-idf term weights as making relevance decisions. *ACM Transactions on Information Systems (TOIS) 26*, 3 (2008), 1–37.

[142] Wu, X., Kumar, V., Quinlan, J. R., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G. J., Ng, A., Liu, B., Philip, S. Y., et al. Top 10 algorithms in data mining. *Knowledge and information systems 14*, 1 (2008), 1–37.

[143] Xin, Q., Behrang, F., Fazzini, M., and Orso, A. Identifying features of Android apps from execution traces. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)* (2019), IEEE, pp. 35–39.

[144] Yang, B., Liping, Z., and Fengrong, Z. A survey on research of code comment. In *Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences* (2019), pp. 45–51.

[145] Zhang, L., Kuljis, J., and Liu, X. Information visualization for DNA microarray data analysis: A critical review. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 38*, 1 (2007), 42–54.

VITA

Rakan Alanazi completed his bachelor's degree in Information technology from Northern Border University (NBU), Saudi Arabia in 2012. Then, he works as a teaching assistant at (NBU). He obtained his M.S. degree in Computer Science Electrical Engineering at the University of Missouri Kansas City in Fall 2016. Rakan starts his research with Dr. Lee in Fall 2018; since then he has published two research papers and one journal. Rakan has developed several software tools namely, Code2Graph, Medl.Ai, and CodEx. His research interests include Software Engineering and Machine Learning.